ELECTANON: A BLOCKCHAIN-BASED, ANONYMOUS, ROBUST AND SCALABLE RANKED-CHOICE VOTING PROTOCOL

by

Ceyhun Onur

B.S., Computer Engineering, Koç University, 2016

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2022

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Arda Yurdakul for her invaluable advice, support, and patience during my thesis study. I am also grateful to my thesis committee members, Prof. Öznur Özkasap and Prof. Can Özturan, for their time and assessment of this thesis. I also would like to thank Kazım Rıfat Özyılmaz for his invaluable suggestions and for sharing his knowledge with me without hesitation. I also would like to thank all the IoT-BC team members for their invaluable feedback and support. I also would like to express my gratitude to all my professors at Boğaziçi University.

I give my deepest thanks to my family for the support, love, and patience they provided me during my study. I found the most needed motivation in even the darkest times with their support. I also would like to offer my most profound gratitude and love to my fiancee, who continuously motivated me during this study.

I also would like to thank Atılberk Çelebi and Bedirhan Çaldır, who have been classmates with me for a long time from undergraduate years in Koç University to my graduate years in Bogaziçi University.

I acknowledge that this thesis research was partially supported by BAP committee under project no: BAP-17A01P7.

A scientific article prepared from this thesis work is under evaluation for publication as the partial fulfillment of the graduation requirements. There may be substantial overlap with this thesis when the article is published. All tables, figures and images in this thesis are prepared by me.

ABSTRACT

ELECTANON: A BLOCKCHAIN-BASED, ANONYMOUS, ROBUST AND SCALABLE RANKED-CHOICE VOTING PROTOCOL

Remote voting has become more critical in recent years, especially after the Covid-19 outbreak. Blockchain technology and its benefits like decentralization, security, and transparency have encouraged remote voting systems to use blockchains. Analysis of existing solutions reveals that anonymity, robustness, and scalability are common problems in blockchain-based election systems. In this thesis, we propose ElectAnon, a blockchain-based, ranked-choice election protocol focusing on anonymity, robustness and scalability. ElectAnon achieves anonymity via zero-knowledge proofs. Robustness is realized by removing the direct control of the authorities in the voting process. Scalability is ensured by treating each ranked-choice ballot as a permutation list, then encoded into a single integer that can be efficiently stored. The proposed protocol includes a candidate proposal system to provide an end-to-end election solution. We also discuss three different extensions in this thesis. The Multiple Elections extension provides a mechanism to use the same set of voters for multiple elections. The Merkle Forest extension minimizes the trust assumption on election authorities in exchange for a decrease in scalability. The Assisted Merkle Tree extension offers just the opposite tradeoff by increasing scalability in favor of requiring external assistance from authorities. ElectAnon is implemented using Ethereum smart contracts and a zero-knowledge gadget, Semaphore. The implementation includes two different sophisticated tallying methods, Borda Count and Tideman. Results show that ElectAnon is capable of running feasibly with up to 100,000 voters and reduces the gas consumption up to 89% compared to previous works.

ÖZET

ELECTANON: BLOKZİNCİR TABANLI, ANONİM, SAĞLAM VE ÖLÇEKLENEBİLİR TERCİHLİ OYLAMA PROTOKOLÜ

Son yıllarda, özellikle Covid-19 salgını sonrasında, uzaktan oylama sistemleri önem kazanmaya başlamıştır. Blokzincir teknolojisinin merkeziyetsizlik, güvenlik ve şeffaflık gibi faydaları, uzaktan seçim sistemlerini de bu teknolojiyi kullanmaya teşvik etmektedir. Mevcut araştırmalar üzerine yapılan analizler; anonimlik, sağlamlık ve ölçeklenebilirliğin blokzincir tabanlı seçim sistemlerinde yaygın sorunlar olduğunu ortaya koymaktadır. Bu tezde, anonimlik, sağlamlık ve ölçeklenebilirliğe odaklanan blokzincir tabanlı, tercihli oylama protokolü olan ElectAnon'u önermekteyiz. Protokol, anonimliği sağlamak için sıfır bilgi ispatlarını kullanır. Protokol, sağlamlığı arttırmak için seçim yetkililerinin seçime doğrudan müdahele etmesini engeller. Protokol, sıralı oy listelerini verimli bir kodlama algoritmasıyla saklayarak ölçeklenebilirliği arttırmayı amaçlar. Ayrıca protokol aday önerme sistemini de içerir. Ayrıca tezin içersinde üç farklı eklenti de ele alınmıştır. Çoklu Seçim eklentisi, aynı seçmen grubunu birden fazla seçimde kullanmak için bir mekanizma sağlar. Merkle Ormanı eklentisi, bir miktar ekstra maliyet karşılığında seçim yetkilileri üzerindeki güven varsayımını en aza indirir. Destekli Merkle Ağacı eklentisi, yetkililerden ek bir yardım gereksinimi karşılığında ölçeklenebilirliği artırmayı hedefler. ElectAnon, Borda Count ve Tideman olmak üzere iki farklı oy sayma yöntemi içerir. ElectAnon, Ethereum akıllı sözleşmeleri ve sıfır bilgi ispat uygulaması olan Semaphore kullanarak geliştirilmiştir. Test sonuçları, önerdiğimiz protokolün 100.000'e kadar seçmenle uygulanabilir bir şekilde çalışabileceğini ve daha önceki çalışmalara göre gaz tüketimini %89'a varan oranlarda azalttığını göstermiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

gen^{ID_i}	Identity generation function
gen^{ZK}	The zero-knowledge key generation function
h	Hash function
n_c	Candidate count
P_i	Zero knowledge proof generated by voter
s_i	Identity random seed

LIST OF ACRONYMS/ABBREVIATIONS

ABI	The Contract Application Binary Interface
CID	Candidate ID
$Circ$	The Zero-Knowledge Circuit
DAO	Decentralized Autonomous Organization
DoS	Denial of Service
EA	Election Authority
EVM	Ethereum Virtual Machine
$ExtN$	External Nullifier
FPTP	First-Past-The-Post
$genProof$	Proof Generation Function
$genWitness$	Witness Generation Function
ID_i	Voter Identity
IDC_i	Voter Identity Commitment
IPFS	The InterPlanetary File System
LT	Lifetime
$MainSC$	The Main Smart Contract
MPE_i	Merkle Path Elements of the Voter
MPI_i	Merkle Path Index of the Voter
MR_C	Merkle Root Generated by the Circuit
MR_S	Merkle Root Stored in the Smart Contract
NH_i	Nullifier Hash of the Voter
$Null_i$	Nullifier of the Voter
$Params$	Final Parameters from Perpetual Power of Tau Ceremony
PoW	Proof of Work
PR	Proportional Representation
$PrivK_i$	Private Key of the Voter
$ProveK$	The Zero-Knowledge Proving Key
$PubK_i$	Public Key of the Voter Identity

RCV	Ranked Choice Voting
SC	The Smart Contract
$Sign_i$	Signature of the Voter
SNARK	Succinct Non-Interactive Argument of Knowledge
$Trap_i$	Voter Random Trapdoor
TS	Tally State Storage Mapping
URL	The Remote Storage Address
UTXO	Unspent Transaction Output
$VerifyK$	The Zero-Knowledge Verifier Key
$VerifierSC$	The Verifier Smart Contract
VH_i	Committed Vote Hash
VID_i	Vote ID of the Voter
VSk_i	Vote Secret Key of the Voter
ZK	Zero-Knowledge

1. INTRODUCTION

The Internet's global success and its adoption in the early 2000s have also affected voting systems. The internet voting, or simply *i-voting*, has enabled voters to cast their ballots remotely, unlike those legacy voting systems which require voters to show up in a specific place like voting booths [1]. The i-voting brings possible advantages like reduced operational costs for elections, time-saving, increased voter participation, and improved transparency in elections [2]. Estonia [3] and Switzerland [4] are two early adopters of *i-voting* in nation-wide government elections. An Estonian governmental agency, Enterprise Estonia (EAS), reported that the i-voting saved approximately 11,000 working days cumulatively in the 2011 Estonian parliamentary election [5]. The same report mentions that the saved costs were roughly equal to 504,000 euros.

Remote systems have become even more significant in recent years, especially after the COVID-19 outbreak. The pandemic has pushed existing systems and procedures to be implemented remotely. Legacy election systems, which oblige voters to go to a specific place to cast a ballot, have also been impacted by the pandemic. For example, almost half of the ballots were cast through mail voting in the US 2020 presidential election [6]. Research shows that internet voting can be a better solution in pandemic periods, as it can give faster results and be more cost-effective than other channels like mail voting [7, 8].

Despite the advantages of internet voting, concerns around Internet-based elections have also been brought into the light. Main concerns toward i-voting revolves around the possibility of *server-side* attacks [2]. The *denial-of-service* (DoS) is a well-known server-side attack type that threatens most online systems. In the DoS attack, the adversary floods targeted servers and services with redundant and resource-heavy requests to overload systems to make them unavailable to respond to legitimate requests. This attack can be widely expanded by distributing and increasing the number of the attack origins. This is also known as *distributed denial-of-service*(DDoS) attack.

DoS attacks are more harmful to centralized systems since it is easier to concentrate the attack on a single target [9]. Another problem with i-voting is that the centralized nature of the Internet makes it more susceptible to censorship by central authorities [10].

Blockchain has emerged as a new paradigm to solve concerns like censorship, targeted server-side attacks, single point of failure using a decentralized infrastructure in contrast to the centralized Internet. Blockchain technology also offers distributed, secure, privacy-preserving, and immutable applications. These innovations of blockchain technology are considered to be very suitable for voting systems as well [11]. A comprehensive review paper shows that the number of research publications in blockchain-based voting has increased from 7 to 30 in 2 years between 2017 and 2019 [12]. We also think that just as blockchains can be useful for voting systems, voting systems can also be very useful for blockchains. Many existing blockchain-based applications, such as decentralized autonomous organizations (DAOs), use voting mechanisms [13]. Thus a solid and efficient voting system can be useful for blockchain-based applications which require voting and elections.

In this work, we analyze and discuss the secure election requirements along with state-of-the-art solutions in blockchain-based election systems. Our analysis of existing solutions has shown us that anonymity, robustness, and scalability are common problems. We find that even though most of the works can achieve privacy, they cannot successfully achieve anonymity because election authorities can know the real identities of voters, and they can distinguish a voter from another. We find this might carry additional risks for blockchain-based elections since addresses in blockchains can be traced, and election authorities can expose a link between voters' blockchain addresses and their real identities. In addition, most of the works use complex encryption and decryption schemes to preserve privacy. We find that these complex schemes are very inefficient for blockchains and cause high voting transaction costs, making existing systems unscalable. Robustness was also not assured as most of the protocols are disruptable in the voting process or require additional assistance from authorities to tally votes.

In the light of these analyses, we propose ElectAnon, a blockchain-based ranked-choice election protocol with additional focus in *anonymity*, *robustness* and *scalability*. In the proposed protocol, we aim to achieve anonymity by using zero-knowledge proofs. ElecAnon minimizes any need for manual handling between election phases and reduces the election assistance in the voting to achieve *robustness*. ElecAnon aims to minimize voting costs to increase scalability by using efficient storage mechanisms and other optimization techniques. ElectAnon targets to achieve *privacy* through *anonymity* with zero-knowledge proofs. This not only brings a novel solution for *privacy* issues, but it also increases the *scalability* by replacing complex encryption schemes with zero-knowledge proof schemes, which are more optimized and efficient for blockchains.

1.1. Our Contributions

In this thesis, we propose a new blockchain-based voting protocol, ElectAnon. Our main contributions in this work are presented as follows:

- We gather known secure election requirements and extend them for secure online elections. *Anonymity* is derived from existing *Privacy* requirement and discussed. The current definition of *Robustness* is expanded by adding *Autonomy* to it. *Scalability* is also redefined and included in blockchain-based election requirements.
- A privacy-preserving, blockchain-based voting protocol, ElectAnon, is proposed. The protocol fulfills the discussed election requirements.
- A scalable ranked-choice voting system is inherited in the protocol to conduct more democratized elections.
- A candidate proposal system is included in the protocol to provide an end-to-end decentralized election experience.
- A modular and algorithm-agnostic mechanism is used in tallying to switch between different methods easily. Two ranked-choice tallying methods, *Borda Count* and *Tideman*, are implemented and analyzed.

- Possible extensions are discussed to optimize the proposed protocol for different use cases. One of these extensions shows a possible mechanism to run multiple elections with the same voters. Another one can be used to adjust trust assumptions and conduct even more *trustless* election environments. We also discussed another extension that can reduce the costs further in favor of increased trust assumption in election authorities.

We used *Ethereum Virtual Machine* (EVM) [14] based smart contracts and an EVM-based zero-knowledge tool called *Semaphore* [15] in our implementations. Our work supports other EVM-based blockchain platforms. ElectAnon can be especially beneficial for decentralized autonomous organizations (DAOs) because of its inherently democratic voting protocol and the ability to work with existing EVM-based smart contracts.

Performance tests are conducted in the *Ethereum* and *Avalanche* local test networks to measure the scalability of the implemented solution. We see that our work is capable of running feasibly up to 100,000 voters. Moreover, we show that our work can scale up the number of voters indefinitely with increased costs. Test results show that our work reduces the total election gas consumption by 89% compared to existing solutions.

1.2. Thesis Organization

The thesis is organized as follows. The following chapter explains the general blockchain and election concepts, along with secure election requirements. Chapter 3 provides a detailed analysis of selected previous works. Chapter 4 starts with a brief preliminary section to explain used algorithms and concepts. The chapter continues with the proposed ElectAnon protocol with its implementation details and provided extensions. Chapter 5 presents the security analysis, test results, and comparison with previous works. Chapter 6 concludes the thesis with an overview of achieved results and gives directions for future research.

2. BACKGROUND

2.1. Blockchain

Even though blockchains have become very popular in recent years, the history of blockchains roots back to the 1970s [16]. There were many different studies conducted on topics like timestamp-able ledgers, e-cash schemes, and digital currencies in peer-to-peer networks from the 1970s to 2008 [16]. Bitcoin whitepaper, the pioneer of blockchain technology, was published in 2008 by an anonymous identity, Satoshi Nakomoto [17]. Bitcoin whitepaper proposes secure, agreeable, and decentralized asset transfers in peer-to-peer networks. These asset transfers are referred to as *transactions* in the whitepaper. Transactions are wrapped in timestamped containers called *blocks*. Blocks are in total order and appended to each other by referencing previous ones. Bitcoin brings Sybil-attack protection called Proof-of-Work (PoW). In PoW, verifier nodes, a.k.a *miners*, compete in a race to create valid blocks by solving a hash-puzzle. The first miner to solve this puzzle gets the right to wrap transactions into a block and propagates them to the network. Each network member, i.e., node, verifies these propagated blocks. Miners get a reward in return if they successfully create valid blocks. Bitcoin also effectively solves *double-spendable* transactions with a new method called unspendable transaction output, i.e *UTXO*. In *UTXO* model, each transaction refers to an existing UTXO. UTXOs are considered spent when they are referred to in transactions. So when they are referred twice, one of the referring transactions is invalidated. A novel consensus model called *Nakomoto Consensus* is also presented in the whitepaper. Participants (nodes) agree to follow the largest chain in this consensus model to reach a network-wide consensus.

Bitcoin brought a groundbreaking approach to asset transfers with modern cryptography techniques. It does not only offer a way to decentralize asset transfers but also reduce transaction costs in comparison to traditional banking. Although Bitcoin offered a breathtaking technology, the drawbacks and bottlenecks have been noticed.

One of the main drawbacks is the long finalization times. Bitcoin aims to produce a block every 10 minutes. It means that each transaction is confirmed within approximately 10 minutes. Bitcoin has some basic scripting to support features like time-locking transactions and multi-owner transactions. However, this has turned out insufficient to support a wide range of application use cases.

2.1.1. Ethereum

Vitalik Buterin proposed *Ethereum* in 2014 to address issues with Bitcoin [14]. Ethereum aims to become a global and decentralized computer that can run many different applications, which are called smart contracts [14]. Ethereum has been a huge success since its release in 2015. Smart contracts are applied in various industries like finance, logistics, insurance, entertainment, and art. The number of smart contracts deployed on the Ethereum network was reported up to 1.5 million in 2020 [18]. Today, it remains the second-largest cryptocurrency, just one place behind Bitcoin.

Ethereum uses a compiled Turing-complete language called *Ethereum Bytecode* for smart contracts. A virtual machine, *Ethereum Virtual Machine* or simply *EVM*, executes each transaction with a given pre-state and outputs a new post-state. It means that every other node running EVM gets the same post-state with the given pre-state so that a consensus can be reached. EVM transactions require computational resources (CPU, storage, memory) to execute these state transitions. These costs can increase with the increased transaction operation complexity. In order to compensate for these required computational resources, each operation requires a constant *gas* cost. It guarantees that the calculation of total transaction gas cost is deterministic. Smart contract functions are simply compositions of different EVM operations, so they all require a gas cost. Gas cost is constant for all Ethereum Virtual Machines at any time; however, the gas must be paid with a real-world value. Ethereum uses ETH as its native token for fees. Each transaction fee is calculated by $gasCost * gasPrice$. The gas price changes with the network activity. Even though the gas cost of transactions is constant, the gas price can change; thus, the actual transaction cost can fluctuate.

There are many different high-level smart contract languages for the Ethereum Bytecode. *Solidity* is one of the most popular and well supported smart-contract languages [19]. Solidity is a C++-like object-oriented language specifically designed for EVM smart contracts. It supports concepts such as inheritance, loadable libraries, custom types, loops, function overloading.

We have used Ethereum and Solidity as our main blockchain platform and smart contract language in our protocol implementation, respectively. This is because they are both well-designed and highly-supported technologies. Moreover, a smart contract implementation in EVM can be used in other EVM-based chains like Avalanche [20], Binance Smart Chain [21], xDai [22] to name a few. Recently Ethereum took a step to specifically support zero-knowledge proofs with the *Istanbul* hard fork (upgrade) [23]. The upgrade has dramatically reduced gas costs of operations in zero-knowledge proof verification. All of these mentioned reasons and the availability of well-supported development tools had led us to choose Ethereum as our main blockchain platform.

2.1.2. Decentralized Autonomous Organizations

Decentralized Autonomous Organization (DAO) is yet another concept brought by Ethereum. DAO can be defined as "an organization that requires no central management" [13]. In DAOs, the organizational decisions are made through proposals and elections. Each proposal defines an organizational operation, i.e., hiring new employees, managing resources, and deciding on feature sets. Proposals are voted, and final decisions are made through elections. In a typical DAO, voting powers are determined by *governance tokens* [13]. These governance tokens are issued via initial coin offerings (ICOs) to investors [13]. Each of these proposals, elections, decisions, and operations are made on the blockchains, so they are transparent, decentralized, and verifiable. DAOs have become very popular since 2018. At the end of 2020, the number of established DAOs was almost 2000. A recent article shows there are more than 700,000 DAO members in September 2021 [24].

2.2. Electoral Systems

In this section, we give a brief explanation and comparison of two electoral systems: *Choose One* and *Ranked Choice*.

2.2.1. Choose One Voting

In plurality voting, or simply *choose-one* voting, voters select a single candidate on the ballot [25]. The choose-one voting is still widely used in today, within electoral systems like first-past-the-post (FPTP) and Proportional Representation (PR) [26,27]. A choose-one voting ballot example can be seen in Figure 2.1. Choose-one is very simple; thus, it is straightforward to understand and applicable for voters. This simplicity brings a restriction for the expressed preferences as voters are only allowed to cast a single vote. This restriction can cause potential problems. Maurice Duverger, a famous political scientist, devised Duverger's law which states that choose-one plurality-rule elections favor the two-party systems [28]. This favor in two-party systems also increases the wasted votes since the losing side might feel ineffective on the election outcome. Wasted votes might also trigger possible *spoiler effects*. Spoiler effects, or *vote splitting*, is the effect of dividing one candidate's potential votes into several similar candidates. It mostly occurs between parties that adopt similar ideologies so that they can attract votes from similar parties. This would eventually favor the election toward the opponent of that ideology. So in some sense, having multiple parties with the same ideology harms that ideology. In the 2000 US Presidential Race, it is debated that Raphael Nade's entrance to the election had split votes from Al Gore because they both share similar ideologies [29]. As a result, choose-one remains susceptible to tactical voting, spoiler effects, and manipulations.

Candidates	Vote
Candidate A	☐
Candidate B	■
Candidate C	☐
Candidate D	☐
Candidate F	☐

Figure 2.1. Choose-one voting ballot example.

2.2.2. Ranked Choice Voting

Ranked Choice Voting (RCV) enables voters to sort their candidate preferences and vote with a sorted list. Contrary to choose-one voting, RCV allows each voter to express their preferences better in an election. An RCV ballot example can be seen in Figure 2.2. RCV reduces the risk of manipulations that exist for the choose-one ballot system. At the same time, it also supports multiple-candidate elections. James Anest discusses the depths of RCV and its potential contributions to conduct more democratic elections [30]. The author states that RCV can encourage candidates to enter elections without having to worry about spoiling votes for other candidates and also helps losers to stay in the future elections. This is due to the fact that the increased expressed preferences reduce the vote splitting risks dramatically. In RCV, a voter can put similar candidates to higher positions, rather than choosing *one of* them. In RCV, it is very likely that each voter's preference makes a difference in the election. RCV encourages voters to participate in elections because it also reduces wasted votes. One of the debates against RCV claims that RCV could be too complicated for voters since it is relatively new when compared to FPTP or PR [31]. However, Anest argues that with the recent developments in electronic voting, it could be practical to use RCV in large-scale elections [30].

Candidates	1st	2nd	3rd	4th	5th
Candidate A	☐	■	☐	☐	☐
Candidate B	■	☐	☐	☐	☐
Candidate C	☐	☐	■	☐	☐
Candidate D	☐	☐	☐	■	☐
Candidate F	☐	☐	☐	☐	■

Figure 2.2. Ranked-choice voting ballot example.

2.3. Secure Election Requirements

There are some requirements that must be satisfied in order to conduct a secure online election. D.A Gritzalis identifies some of these requirements in his work in 2002 [32]. Gritzalis' identified requirements are still valid even today. A recent work [33] in 2021 mentions a similar set of requirements for secure blockchain-based online elections. We gathered and presented existing definitions of *Eligibility, Uniqueness, Privacy, Fairness, Soundness, Universal Verifiability, Individual Verifiability* secure election requirements. We also expanded existing definitions of *Privacy* and *Robustness*. We also introduce *Scalability* as a secure online election requirement.

2.3.1. Eligibility

Only eligible voters must be able to cast ballots in an election [32]. The eligibility of a voter is generally decided by some set of rules. Some of these rules may be listed as being a resident in a particular state, being old enough to vote, having no criminal records. In most cases, election authorities collect some documents like ID cards or biometric data to decide on a voter's eligibility [34].

2.3.2. Uniqueness

Uniqueness or *Unreusability* ensures that no voter can cast more than one ballot in an election. In other words each voter can vote only once [32, 35].

2.3.3. Fairness

No intermediate results should be available to be obtained. In other words, voters should not be able to alter their preferences according to the intermediate results [32, 33].

2.3.4. Soundness

Only valid ballots should be taken into account in the tallying process. In other words, invalid ballots should be discarded and not be tallied [33].

2.3.5. Universal Verifiability

The fairness and the correctness of an election result must be verifiable universally. Even non-participants must be able to validate the election result [36, 37].

2.3.6. Individual Verifiability

Voters must be able to verify that their ballots are cast correctly in the election [36, 37]. We also think that voters should be able to verify all their interactions with the election protocol, especially in elections that have multiple states and phases.

2.3.7. Privacy and Anonymity

Votes should not be distinguishable in terms of who cast them. It also means that no one should be able to figure out which voter used which ballot [36].

Online voting, especially repeated elections in public networks, is susceptible to linking voting preferences to voters' digital identity. This might include linking a digital anonymous identity to the actual identity via *linkage attacks* [38]. Because of this link, the privacy of voters can be reduced. This could carry a higher risk in the case that a party, like the election authority, knows the actual identities. We expand the *Privacy* requirement by ensuring that no operation in an election can be linked to actual identities. We call this *Anonymity* of voters. If the *Anonymity* is guaranteed, then no parties, even election authorities, can know who interacted with the election protocol.

2.3.8. Robustness

Robustness is another crucial requirement for voting systems. This condition ensures the inability of any parties to disrupt an ongoing election [33]. Robustness is extended in the work [39] by adding *tallying availability* on top of *voting availability*. Voting availability ensures that eligible voters can finish the voting process without any disruption. Tallying availability ensures that valid votes can be tallied correctly without any disruption. In other words, tallying availability guarantees that cast votes will not be lost in vain. This is crucial for blockchain systems since each transaction requires a transaction fee and cost. If a disruption happens after vote casting, it means that this cost would be for nothing. Most of internet-voting protocols typically consist of different phases like *Initialization*, *Voting* and *Tallying* [2] We extend *Robustness* requirement by adding *autonomy*, which ensures that there can be no halting/freezing between phase changes.

2.3.9. Self-Tallying

Everyone should be able to calculate the election result and come up with the same result as others [37]. This ensures there is no need for authorities or any specific actors to calculate the election result. Hence everyone, even non-participants, can learn the election result without depending on other parties.

2.3.10. Scalability

This one is not a direct election requirement but one of the crucial requirements for generic electronic systems. We define *Scalability* as the maximum number of voters and candidate counts in an election. Most blockchain platforms have a fixed capacity of transaction and block size. It means that the *Scalability* in those systems are more evident and can be easily measurable when compared to traditional centralized systems. There are also other considerations that should be taken into account for a scalable election system, like required computational resources and calculation times.

3. RELATED WORKS

In this section, we analyze and discuss existing blockchain-based e-voting studies. A comprehensive work *Trends in Blockchain-Based Electronic Voting Systems* [40] evaluates and scores more than 50 blockchain-based e-voting studies. Score evaluation is based on nine different qualification criteria questions and scores each study with the number of criteria it provides. We picked some of these studies to do a more detailed review and comparison as follows:

- McCorry et al. [41] scores a %100 in [40] and also one of the earliest (2017) blockchain based works.
- Chaintegrity [36] scores also %100 and has a smart contract implementation.
- Yang et al. [35] has a %89 score, and also uses a ranked-choice electoral system like ours.
- Panja et al [42] also scores %89 and uses a smart-contract based borda-count voting with gas consumption measurements.
- PriScore [37] is not scored in the work [40], however worth to be included as it uses *score-voting* and has a smart contract implementation.

We evaluated these works in the following aspects: *Eligibility, Uniqueness, Privacy, Anonymity, Fairness, Soundness, Universal Verifiability, Individual Verifiability, Robustness* and *Scalability.* Table 3.1 categorizes and summarizes this evaluation.

Table 3.1. Evaluation table of selected blockchain based e-voting studies.

Work	Election requirement										Scalable	Electoral System	Platform
	E	U	P	A	F	S	UV	IV	R	S-T			
McCorry et al. [41]	✓	x	✓	x	✓	✓	✓	✓	o	✓	x	Yes-No	Public/Ethereum
Chaintegrity [36]	✓	✓	✓	x	x	✓	✓	✓	x	x	x	Choose-One	Abstract
Yang et al. [35]	✓	✓	✓	x	o	✓	o	✓	x	o	x	Ranked Choice	Abstract
Panja et al. [42]	✓	x	✓	x	✓	✓	✓	✓	o	✓	x	Ranked-Choice	Public/Ethereum
Priscore [37]	✓	o	o	x	✓	✓	✓	✓	o	o	x	Ranked-Choice	Public/Ethereum
ElectAnon (this work)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Ranked-Choice	Public/Ethereum

E: *Eligibility*, U: *Uniqueness*, P: *Privacy*, A: *Anonymity*, F: *Fairness*, S: *Soundness*, UV: *Universal-Verifiability* IV: *Individual-Verifiability*, R: *Robustness*, S-T: *Self-Tallying*
✓: implemented, x: not implemented, o: partially implemented

3.1. Comparative Analysis

3.1.1. McCorry et al.

McCorry et al. [41] implements the *Open-Vote Network(OV-net)* protocol originally proposed by Hao et al. [43]. McCorry et al. use a blockchain-based smart contract system to implement the OV-net protocol. The protocol uses shared-key encryption to encrypt individual votes. Tallying can be done by combining encrypted votes and obtaining the decrypted result without actually decrypting individual votes. The study accomplishes *Eligibility* by publishing a list of eligible voter accounts in the smart contract. *Uniqueness* is not mentioned in work. We suppose that an eligible voter can double vote using a different voting key. Hao et al.'s original OV-net protocol [43] mentions that other participants can notice this type of action, but it does not mention any countermeasure.

Privacy is ensured in work since preferences are kept encrypted with the shared-key encryption and cannot be decrypted without the voter's secret key. However, the work does not guarantee *Anonymity* since election authorities need to register eligible voters with their blockchain addresses. This means that election authorities can link the real identities of eligible voters with their blockchain addresses. When voters interact

with the protocol, election authorities can distinguish the actual identities of voters. *Fairness* was an open-question in the original OV-Net protocol [43].

The very last voters in the election held the last piece of the shared key; thus, they can self-tally the election before revealing their own key. McCorry et al.'s work [41] fixes this by adding an additional step that locks (commits) every voter's preferences before tallying. *Soundness* is achieved through verifying each ballot with Zero-Knowledge proofs. *Universal & Individual Verifiability* are achieved through *self-tallying* and via verifiable blockchain transactions. We consider *Robustness* is partially implemented here since last voters have the ability to disrupt the whole election by not providing their shared keys. This is partially solved by adding a mechanism that requires voters to deposit some tokens beforehand and refund when they successfully cast their votes. However, voters with harmful intentions can still disrupt the whole voting process despite the economic incentives.

The work is also cannot be considered as *Scalable*. At the time of their writing, they hit an Ethereum block capacity (2 million gas) for a single vote in a 50 voter setup. Currently, the block capacity is 30 million gas on average, which makes the work support around 650 voters at maximum as of today. The cost of casting a single vote is shown as approximately 3,300,000 gas which would be equivalent to almost 900$, which is not feasible. The work uses a single *yes/no* based election. This is also very restrictive on the expression of preferences, as mentioned in Section 2.2.1.

3.1.2. Chaintegrity

Chaintegrity [36] implements an e-voting protocol with smart contracts. The underlying protocol uses blind signatures and multiple-round of message exchanges between election authorities and voters. The work provides *Eligibility* by publishing a list of eligible public keys and verifying each signature in the vote registering. *Uniqueness* is also accomplished by marking public keys once they successfully cast a vote.

Privacy of underlying protocol depends on the assumption that voters use different blockchain addresses in different phases. In the protocol, voters register a blinded ballot with their public keys and blockchain addresses; this reveals a link between their public keys and blockchain addresses. We think *Anonymity* is violated here since the election authority knows the public key of voters in registration. Thus, the authority can link blockchain addresses with the real identity of voters. Even though *Anonymity* is not achieved, the work achieves *Privacy* by using homomorphic encryption to tally votes without decrypting them.

In the work, election authorities hold a shared secret key. The authors state that their work meets the *Fairness* requirement under the assumption that the number of honest election holders is above a security threshold. We think that this assumption is too broad since the number of election holders is relatively small when compared to voters. In other words, a subset of election holders can organize a collaborated attack to tally the partial results before ballot casting ends. Thus intermediate results can be obtained by the malicious election holders during the voting procedure, so *Fairness* is violated. *Soundness* is achieved by verifying ballot validity with zero-knowledge proofs. *Universal Verifiability* is achievable at the end of the election by self-tallying the encrypted results and verifying that decrypted results are indeed intact. *Individual Verifiability* is available via verifying blockchain transactions.

Voters need to send their encrypted ballots to election holders to obtain valid signatures. It is possible that the election holder can stall the voting procedure by not providing a signature or providing an invalid one. *Voting verifiability* is violated since this disruption may occur during ongoing voting. The *tallying verifiability* is also violated since election holders have to collaborate together for decrypting the encrypted vote results at the end of the election. It means they can stall the procedure or completely disrupt the election by not providing their keys. Thus, *Robustness* is violated in terms of both *voting verifiability* and *tallying verifiability*. *Self-Tallying* is also not possible since results have to be decrypted and tallied by election holders.

The authors mention that their protocol is scalable since their communication cost is linear in the number of voters ($O(n)$). However, their protocol requires additional steps. Firstly, voters need to send blinded ballots to the smart contract. Secondly, election holders need to receive these blinded ballots from the smart contract, then sign them and send them back to the smart contract. Finally, voters need to resend an unblinded version of their ballots. In total, the voting procedure requires three transactions to blockchain for a single vote. Tallying also requires one transaction. Each additional transaction reduces scalability. Our work requires only two transactions for voting and no transaction for tallying. It means that this work requires one additional transaction for each vote and one for tallying when compared to our work. Any additional transactions, especially for voting, might end up increasing costs in a large-scale election. They use plurality i.e choose-one voting [25]. The work does not mention a specific blockchain platform for their implementation.

3.1.3. Yang et al.

Yang et al. [35] uses ElGamal scheme together with a new group-based encryption proposed in work. The work implements *Eligibility* by registering eligible public keys and verifying signatures in the vote casting. *Uniqueness* is achieved by storing the digital signatures in the blockchain and checking that no signature is used more than once. *Privacy* is ensured by encrypting votes twice. Voters encrypt their preferences with a voting key and also with a public candidate key. So, in the end, individual votes can only be decrypted by both the candidate's private key and the voter's private voting key. During eligible public keys registration, the election authority knows the real identities of voters, so the election authority knows the link between the public keys of voters and their identities. Voters use the same public keys and blockchain addresses in the vote casting; thus, election authority can deduct their real identities when they cast their votes; this violates *Anonymity*. Candidates can decrypt their own results before revealing their secret keys at the end of the *Voting* state. Thus each candidate will know their results before the result is publicly announced. Since this can occur only after the voting phase, we consider *Fairness* is partially implemented.

Soundness is guaranteed by zero-knowledge range proofs [35]. *Universal Verifiability* is partially available since the result must be decrypted by the candidate keys before the result can be verified. *Individual Verifiability* is available through verifying blockchain transactions.

Robustness is not guaranteed since candidates have to reveal their secret keys in order to tally the results. This means that candidates can mask their results by not revealing their keys. Authors mention that this can be detectable and punishable by giving these malicious candidates a 0 score. However, this clearly ignores preferences of used votes for these candidates and disrupts the election. Since it is not possible to tally the results without candidate keys, we consider *Self Tallying* is partially implemented in this work. Voters need to encrypt each candidate in their ballots with their related candidate secret keys. This creates an additional overhead with the increased candidate count. Voters also have to form a zero-knowledge proof for their votes. In their performance analysis, they measured a 100 KB total size for one vote with 15 candidates. This does not fit into one Ethereum block, which is almost 80 KB on average at the time of writing this paper [44]. Because of these reasons, we think that the study cannot be considered as *Scalable*. The study uses *ranked-choice voting*. The research does not mention anything about blockchain platform choices.

3.1.4. Panja et al.

Panja et al. [42] add a ranked-choice election scheme to the existing work of McCorry et al. [41], and OV-net [43]. The work inherits the same election requirements from the previous work of McCorry et al. [41]. The only differences are *Scalability* and *Electoral System*. Unfortunately, adding a ranked-choice electoral scheme to the existing work [41] has made it even less scalable. The communication & computation costs are shown as $O(nk^2)$ where n is voter count and k is candidate count. They run their implementation with 80 voters and with five candidates in Ethereum. They mention that even a single voting cost exceeds the block limit of 8 million. They tried to split the voting transaction into five sub-transactions, but in the end, a single voting

transaction costs 40,102,222 gas. Unfortunately, this result is even more expensive than the original work of McCorry et al. [41].

3.1.5. PriScore

PriScore [37] uses a *distributed ElGamal* scheme and *zero-knowledge proofs* [45]. The work achieves *Eligibility* by publishing a list of eligible voter public keys and then verifying these public keys in the voting phase with zero-knowledge proofs. The work can detect when a voter tries to cast multiple votes but does not mention any countermeasures against it. So we suppose *Uniqueness* is partially supported. The work uses a similar *distributed secret key encryption* scheme like McCorry et al. [41] and Panja et al. [42]. As mentioned before, in these schemes, if any voter abandons their committed vote, it can disrupt the whole voting. This work also suffers from the same issue. In order to solve this, they added a commitment mechanism so that they can terminate the whole election in case of a registered voter does not commit a vote. This solution can only work for the abandoning vote problem in the commit phase. It is still possible that a committed vote can be abandoned before the reveal, thus can disrupt the election. The work has two tallying algorithms for two different cases. In the first case, every registered voter casts their ballots, and thus everyone can self-tally the result via the *distributed ElGamal* scheme. However, in the second case, where abandonment happens, remaining voters need to collaborate together to re-calculate the result from the remaining votes. The study mentions that the second algorithm requires voters' secret keys as inputs to the algorithm. This clearly violates the *Privacy*. We assume that *Privacy* is partially implemented since the second situation, where voters abandon their votes, is an edge-case. The election authority collects public keys from voters and generates a list of eligible voters. Eligible voters provide their public keys in *Commit* state to verify that they are on the eligible list. Since the Election authority knows their actual identity and their linked public key, *Anonymity* is not guaranteed.

Fairness is guaranteed with the fact that encrypted votes cannot be decrypted until the end of the tallying process. *Soundness* is achieved by ensuring the ballot

validity with zero-knowledge proofs. *Individual Verifiability* satisfied by verified signatures on ballots; thus, each voter can check if their votes are published and tallied correctly on the blockchain. *Universal Verifiability* is ensured by homomorphic addition in ElGamal encryption. *Robustness* is partially violated since abandoned votes can disrupt the election. The work mentions that the tallying algorithm can tolerate at most one abandoned vote. However, it is also mentioned that an abandoned vote is not recoverable during the commit phase. So if one voter does not participate in the commit phase, the whole election needs to be reconstructed. We also think that *Self-Tallying* is partially supported here since voters cannot tally the result without collaborating together in the abandoned vote case.

The authors of the work provide a gas consumption table of different zero-knowledge verification operations. We calculated the total gas consumption of each state with ten candidates and 50 voters. *Setup* state is not mentioned for gas consumption, so we skipped it. *Commit* state consumes a total of almost 1,100,000 gas. *Vote* state consumes nearly 3,500,000 gas. Commit and Vote phase consumption increases linearly with respect to candidate count. The work uses an efficient *Self-Tally* algorithm, and it consumes almost 60,000 gas. The abandoned case tallying algorithm consumes nearly 500,000 gas which increases linearly with respect to both candidate and voter counts. The authors did not specify how voters can collaborate for tallying in the abandoned case. We assume this might introduce additional overhead to the network. In short, we assume that the work cannot be considered as *Scalable* since a single vote casting consumes almost 4,500,000 gas which is almost half of the block gas capacity. We also think that the gas consumption of the abandoned tally function will hit Ethereum's block size of 30,000,000 gas in an election with ten candidates and 25000 voters. The work uses *ranked-choice* electoral system with Ethereum as their platform choice.

3.2. Discussion

We have iterated over some most qualified works mentioned in work *Trends in Blockchain-Based Electronic Voting Systems* [40]. A summary table can be found in Table 3.1. We have identified three common problems in the prior works. These are *Anonymity*, *Robustness* and *Scalability*.

Anonymity was not considered thoroughly in the related works. Prior studies try to accomplish *Privacy* by keeping votes encrypted and secret. This can hide the link between a voter and their actual preference. In other words, they aim to secure the *vote* side of the vote-voter link instead of the *voter* side. We aim to secure the *voter* side of the link in our work. The prior works fail to accomplish this because election authorities hold the information of voters' real identities and their public keys or blockchain addresses. This is needed in order to form a list of eligible voters properly, so *Eligibility* can be reassured. However, when voters act in an election (i.e., vote, commit, register, etc.), election authorities will know who acts on which function. The *Anonymity* becomes more critical if the same list of eligible voters is being used many times. Election authorities may form and enhance a link between the voter and the vote usage pattern in every repeated election. This can be even more apparent for studies like Yang et al. [35], McCorry et al. [41], Priscore [37]; since they need to repeat their elections in case of abandoned votes. Most blockchain addresses are pseudo-anonym by design [46]. It means addresses can be tracked down to affiliate the user with their chain activities. In blockchain-based voting systems, it can lead to the exposure of a link between voter identities and different blockchain activities. These blockchain activities include transactions, token balances, ownerships, and relations with other addresses. If *Anonymity* is not guaranteed, then election authorities can link actual identities to these blockchain activities.

The *Robustness* also could not be achieved properly in prior works. Yang et al. [35] and Chaintegrity [36] depend on third-party actors to decrypt ballots in tallying. This clearly violates as these third-party actors can disrupt the election by providing invalid

credentials or not providing credentials at all. Some other works like McCorry et al. [41], Yang et al. [35] and PriScore [37] use shared secret-key encryption, and thus susceptible to vote to abandon. McCorry et al. [41] tries to solve this by incentive mechanisms, by depositing and refunding some assets. However, if the political advantages of an election exceed the incentives, voters can still disrupt the voting. PriScore [37] tries to solve this with additional cryptographic operations; however, it also fails to apply it for more than one abandoned vote. Every blockchain transaction costs additional fees. So a large-scale election setup can be very costly, especially the cases like nationwide elections. As a result, *Robustness* becomes a very crucial requirement in e-voting, especially for blockchains. In our work, we tried to minimize the risk of ongoing election disruption and also eliminated the possibility of election repetition.

Scalability is another aspect that could not be achieved properly in prior works. They either fail to provide feasible costs or cannot conduct large-scale elections. We think that this is because most of the works tried to preserve *privacy* with complex encryption/decryption schemes like homomorphic encryption, ElGamal encryption, and shared key encryptions. We think these are inefficient for blockchains, especially for Ethereum based platforms. Works like McCorry et al. [41], Panja et al. [42] and Priscore [37] uses Ethereum as the blockchain platform, and they were able to provide gas consumptions of their implementations. Gas consumption is a very strong indicator of scalability since there is a limit on the maximum consumable gas in Ethereum. This limit dynamically changes with the network state and Ethereum fork version. We tried to evaluate their work in today's Ethereum gas limit standards. These works either hit a maximum gas limit with a relatively small voter set or consume too much gas, making them practically impossible. We tried to minimize the voter-side gas consumption in our work so that it can be practically possible to conduct a large-scale election in Ethereum. We think measuring the transaction delays and finalization times is not very reasonable since these completely depend on the network state and the underlying blockchain platform. It could be misleading to measure the scalability of a smart contract, or generally a decentralized application, with transaction and finalization times.

4. PROPOSED SOLUTION

We propose ElectAnon, a privacy-preserving, anonymous, scalable voting protocol. In this section, we provide some background information about incorporated concepts and algorithms before explaining the implementation in detail.

4.1. Preliminaries

In this section, we provide some background information of the algorithms and techniques used in our proposed solution.

4.1.1. Zero-Knowledge Proofs and zk-SNARKs

Zero-Knowledge proofs provide a way to prove the existence of the knowledge without revealing the knowledge itself. There are two main actors in the protocol. The *prover* forms the zero-knowledge proof and the *verifier* verifies the proof. The zero-knowledge term was first mentioned in *The Knowledge Complexity Of Interactive Proof Systems* by Goldwasser, Micali & Rackoff in 1989 [47]. Use cases include proving private data, anonymous authorizations, private payments, computation offloading, and electronic voting [48,49]. For example, one can prove they hold a sufficient amount of balance in their bank account without actually revealing the exact balance itself. Another example could be proving identity is indeed in a given eligible list without actually revealing the identity in order to preserve privacy. The zero-knowledge proofs can also be used to scale-up blockchains by offloading heavy computations to off-chain (i.e., local) while verifying proofs and results on-chain [48].

The earlier zero-knowledge protocols make use of interactive protocols which require verifiers and provers to exchange messages in several rounds. *Zero-Knowledge Succinct Non-Interactive Argument of Knowledge* or simply *zk-SNARK* brought an efficient zero-knowledge protocol that reduces the number of rounds to verify proofs [48].

zk-SNARK proofs are also *succinct* as they can be verified in milliseconds, and their proof sizes can be as small as a few hundred bytes long.

zk-SNARKs are widely used in blockchains both to preserve privacy and to offload heavy computations for scalability. *Zcash* is a well-known blockchain platform that uses zk-SNARKs [50]. Zcash offers full-private blockchain transactions that completely hide sender and receiver addresses and transaction amounts. The success of Zcash has led other blockchain platforms to use zero-knowledge proofs and zk-SNARKs. For example, Aztec [51] and StarkNet [52] are two side networks being developed with zk-SNARKs to off-load Ethereum transactions.

zk-SNARK proofs are constructed with complex mathematical equations. These equations, or constraints, can be in the form of quadratic, linear, or constant equations. Each of these constraints is combined together to form a single equation. These arithmetic equations are formed into circuits [53]. There are some zk-SNARK circuit compilers that abstract away these mathematical equations and generate zk-SNARK constraints by using a higher-level language. Circom [54] and Zokrates [55] are two popular compilers for zk-SNARK circuits. They support some basic software-language concepts like variables, functions, control flows, operators and include types like integers, booleans, arrays, and structs. It's also possible to define private and public inputs and output signals with these languages. They can also generate smart contract verifiers so that proofs can be verified on-chain. Typical steps for generating and verifying zk-SNARK proofs are listed as follows:

(i) Designing a high-level circuit, i.e., writing the logic for the zero-knowledge computation.
(ii) Compilation of the high-level circuit into a set of low-level arithmetic equations.
(iii) Generating verification key and a proving key with the compiled circuit and a secret parameter. Both keys are announced publicly.
(iv) Computation of a secret called to witness by executing compiled circuit with the given public and private inputs.

(v) The proof generation with the witness and the proving key.
(vi) Sending generated proof to the verifier.
(vii) Verification of the proof with the verification key by the verifier.

4.1.2. Semaphore

The Semaphore [56] is a zero-knowledge protocol and set of tools for anonymous signaling. It uses a circuit to prove these three properties:

(i) An identity is eligible to broadcast a signal.
(ii) The signal truly belongs to the identity owner
(iii) The signal is not broadcasted twice.

The Semaphore aims to prove these properties without revealing any extractable information about the user identity. As a result, users can broadcast various signals without revealing their actual identities. A valid proof verifies that the user is indeed on the eligible list. Eligible lists are defined with Merkle trees, so identity verification can be efficiently done with Merkle proofs. The Semaphore contains a *Circom* zk-SNARK circuit and two smart contracts. One of the smart contracts includes a Merkle tree implementation so that smart contract owners can register eligible identities to Merkle trees through the smart contract. The other smart contract verifies zero-knowledge proofs and prevents double-signaling by storing nullifiers. The project also provides a Javascript library *libsemaphore* [57] to seamlessly interact with smart contracts and to generate identities, witnesses, signals, and proofs. More details can be found in the comprehensive article: *Semaphore: Zero-Knowledge Signaling on Ethereum* [15].

Semaphore can be used in a wide range of use cases like anonymous transactions in ZCash, anonymous authentication, mixers, and private voting. We adopted and used the Semaphore in our voting protocol. ElectAnon uses a modified version of Semaphore Merkle trees to reduce the huge cost of voter registration. We explained the modified version in Subsection 4.3.5.

zk-SNARK circuits use a random secret parameter called *toxic waste* in the setup phase to generate proving and verifying keys [58]. This toxic waste must be safely discarded, as it can be used to generate a fake proof. Semaphore uses a ceremony called *Perpetual Powers of Tau* to make sure the toxic waste is discarded [59]. The ceremony guarantees security even if only one participant safely discards their toxic waste and responds to the ceremony. There are two phases in the ceremony. Each phase consists of several rounds. Phase-1 is a common ceremony for generic zk-SNARK circuits [59]. Phase-2 must be applied by each different circuit implementation. The ceremony can continue as long as there are participants willing to participate. At the time of writing this work, there were 71 participants in the repository for Phase-1 [60]. The authors of Semaphore have worked on a follow-up Phase-2 ceremony specifically for the Semaphore circuit [61]. ElectAnon uses the same Semaphore circuit, so we can use the same Phase-2 ceremony in our implementation.

4.1.3. Ranking and Unranking Permutations in Linear Time

Myrvold & Ruskey developed an efficient algorithm to rank/unranked permutation lists [62]. The ranking algorithm takes a permutation list (π), size of the list (n), and its inverse (π^{-1}); then outputs the corresponding rank integer (r). Each rank integer represents a permutation list, since there can be at most $n!$ different permutations, the rank integer (r) can be in the range of between $[0...n! - 1]$. The unrank algorithm takes n, r, and an identity permutation π' and outputs the corresponding permutation list π. Both ranking and unranking algorithms have $O(n)$ arithmetic operation complexity, which makes them practical enough to compute. An example for $n = 3$ can be found in Table 4.1.

We used a ranked-choice electoral system in our voting protocol. In the ranked-choice system, voters from a list of candidates by ranking them from most wanted to least wanted. So each vote list, or *preference list*, is a permutation of the candidate list. Our candidate list is a set of candidate IDs ranging from 0 to candidate count (n_c). Each preference list is required in tallying phase for complex tallying algorithms

like Tideman [63]. We initially tried to store all preference lists as arrays in the smart contract. In the best case, there is only 1 reference list that needs to be stored if every voter votes for the same preference list. However, in the worst case, where every voter votes for a different list, there are $P_n = n!$ different preference lists. Unfortunately, storing arrays in smart contracts is not gas efficient. We developed a solution to efficiently store these preference lists in smart contracts. We used the aforementioned rank/unrank algorithms to encode/decode our preference lists. Instead of storing the whole list, we were able to compute the rank of each preference list and store the corresponding integer rank instead of arrays. Voters compute ranks of their preference lists off-chain(offline) in the voting phase, so it also offloads the computation. Then our tallying function in the smart contract unranks each of these rank integers and computes preference lists to obtain results. Since there can be at most $n!$ permutations, the maximum rankID can be $n! - 1$. So we can easily detect an invalid rankID, or voteID, by checking if the given rankID is in the range of $[0, n! - 1]$. These rank & unrank algorithms do not only reduce the storage cost but also provide us a way to easily detect invalid votes to ensure *Soundness*.

Table 4.1. Example rank/unrank table for n=3.

Permutation	Rank
[1,2,0]	0
[2,0,1]	1
[1,0,2]	2
[2,1,0]	3
[0,2,1]	4
[0,1,2]	5

4.2. System Model

ElectAnon is a self-tallying, privacy-preserving e-voting protocol implemented with EVM smart contracts. The protocol aims to support all of the requirements men-

tioned in Section 2.3. The protocol utilizes the *ranked-choice* electoral system. There are three system actors defined in the protocol. These are *Election Authority*, *Proposers*, and *Voters*. The protocol is composed of six different phases: *Setup, Register, Proposal, Commit, Reveal*, and *Completed*. These phases are consecutive, and together they form a timeline formation. Different actors take different actions in each phase. An overview of our protocol flow and the timeline is presented in Figure 4.1.

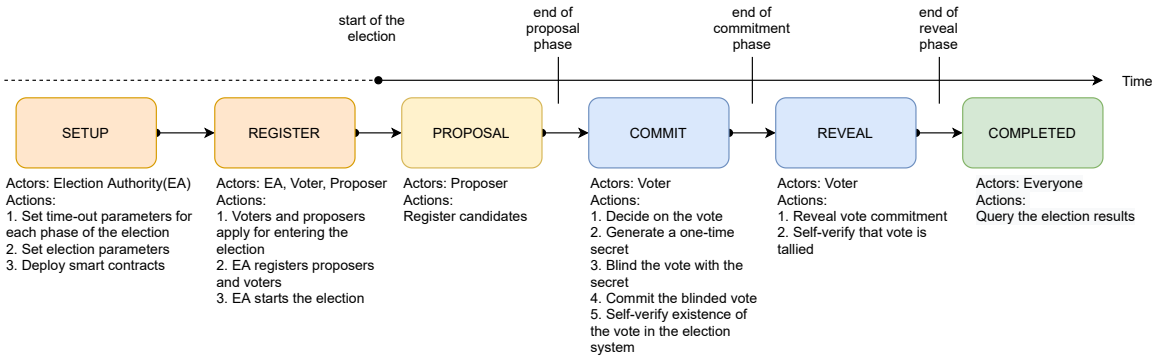

Figure 4.1. ElectAnon Timeline.

4.2.1. System Actors

- *Election Authority* (EA) is responsible for system setup and initialization. EA decides on system parameters and initializes the election by deploying the smart contract to the blockchain. EA is also responsible for registering eligible *proposers* and *voters* to the system. EA role can be assigned to several parties. For example, one party can be solely responsible for the setup/deploying system to the blockchain, whereas another party can register eligible voters and proposers. For the sake of simplicity, these parties are unified under a single Election Authority in this work. EA capabilities and responsibilities are limited. EA does not actively take place in the protocol after the end of the *Register* state.
- *Proposers* can propose their answers for the election question. Each proposed answer, i.e., a proposal, becomes a candidate in the election. Each proposer can only register a single proposal. Proposals are in the form of arbitrary strings.

Proposals are registered only during the *Proposal* state.

- *Voters* can vote for candidates, during the *Commit* state. Voters also participate in the *Reveal* state to reveal their committed votes.

4.2.2. States

ElectAnon protocol implements a timed-transition state machine. The states are *Setup*, *Register*, *Proposal*, *Commit* and *Reveal, and Completed*. Each state has a timed-transition guard, meaning that there is a lifetime/deadline for these states. If any state-specific transaction (function call) is received after the related state, the transaction is rejected. For example, let's say the proposal lifetime is selected as 30 blocks. The state deadline is computed by adding 30 (lifetime) to the block number of the proposal state's start. After 30 blocks, any call to the *Proposal* state-specific function is rejected. In the same example, users (voters) can start using functions defined for the *Commit* state after 30 blocks. These lifetimes are specific for each state and can be defined in the smart contract deployment.

In addition to timed-transition guards (lifetimes), there are also conditional transitions that check if states can transit the next one safely. These conditional transitions mostly check if all actors have taken their actions successfully. For example, if all registered voters commit their votes, *Commit* state switches to the next state, i.e., to *Reveal*. Conditional transitions and timed transitions are used in conjunction; whichever occurs first changes the state to the next one. For instance, even if some of the voters abandon their committed votes in *Reveal* phase, the timed-transition condition can change the state to the next phase if the *Reveal* lifetime is exceeded. This mechanism ensures that the election protocol can continue as usual in an autonomous way without requiring any additional triggering. Figure 4.2 shows the state diagram and transition conditions.

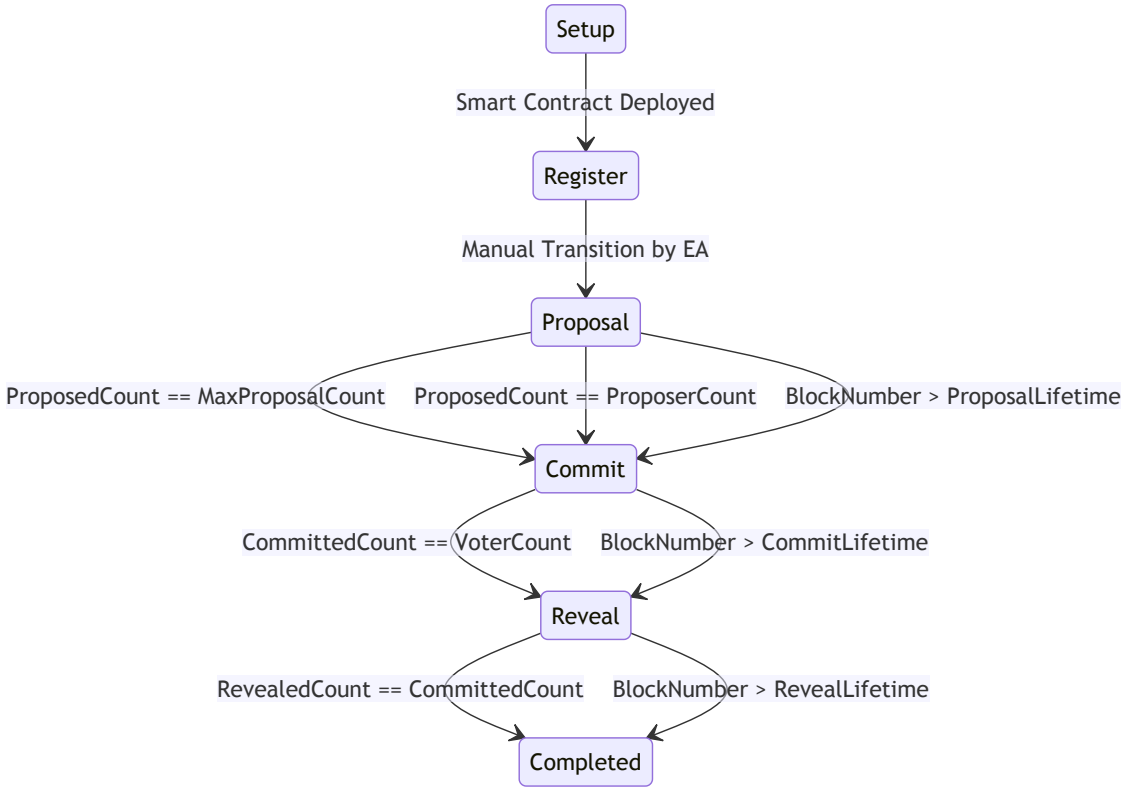

Figure 4.2. ElectAnon State Diagram.

4.2.2.1. Setup State. Election Authority EA initializes the election parameters and protocol at this state. EA decides on the election question at this state. The election question, or poll question, is an arbitrary string that can represent various question topics like "who should be the president" in a presidential election; "which platform should be used" in organizational decisions.

EA prepares the zero-knowledge circuit *Circ* in this state. The Semaphore circuit defines a static *tree-level* inside the circuit code which defines the maximum depth of the Merkle tree. The parameter also indicates the maximum voter count, as voters are registered to the election through the Merkle tree. The maximum voter count equals the Merkle tree leaf count, which is $2^{tree\ level}$. EA can alter the *tree level* in the circuit to change the Merkle tree level and thus the maximum voter count. The Semaphore circuit uses 20 levels by default, which equals to a maximum $2^{20} = 1,048,576$ number of voters in the Merkle tree.

EA announces the zero-knowledge verification ($VerifyK$) and the proving keys ($ProveK$) in this state. These keys are generated from the given circuit $Circ$ and final parameters ($Params$) of *Perpetual Power of Tau* ceremony as

$$gen^{ZK}(Params, Circ) \rightarrow (VerifyK, ProveK).$$

EA can store these keys, and the circuit in a place where every voter can access, preferably a decentralized storage like *The InterPlanetary File System* (IPFS) [64]. The storage address (URL) can be put in the smart contract so that voters can fetch keys to generate their proofs without a need for recalculation of these keys. This step can be fully omitted since each voter can generate these keys on their own, using publicly announced ($Params$) from the Perpetual Power of Tau ceremony.

EA also generates the verifier smart contract ($VerifierSC$) with the verifier key $VerifyK$. The verifier smart contract is embedded (inherited) in the main smart contract ($MainSC$) so that these two can be combined and deployed as a single, smart contract (SC). Later on, EA decides on the election parameters to conduct a custom election. These parameters are *tree level, maximum candidate count, proposal lifetime, commit lifetime*, and *reveal lifetime*. The *tree-level* should be the same as the one in the circuit $Circ$. The *maximum candidate count* indicates the maximum number of proposals that can be registered as candidates. Each *lifetime* (LT) parameter defines the lifespan of the related state in terms of block number, i.e *proposal lifetime* indicates the lifespan of the *proposal* state. EA can deploy the smart contract (SC) with these parameters at this point. This phase is completed when the contract is deployed. Figure 4.3 shows the sequence diagram for this state.

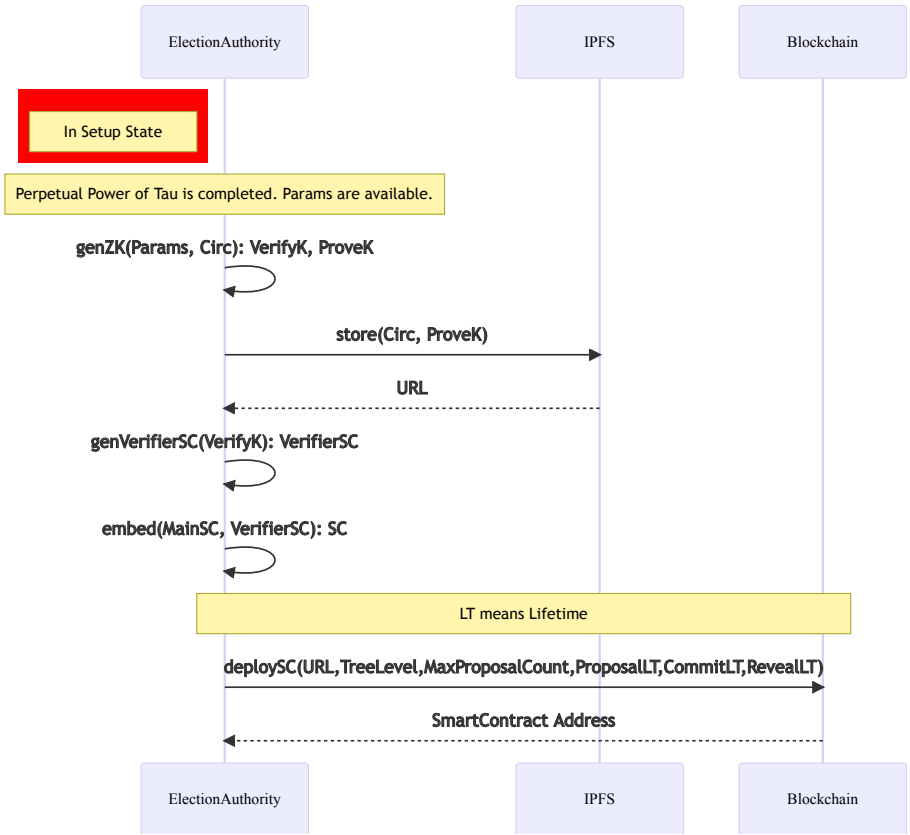

Figure 4.3. Setup State Sequence Diagram.

4.2.2.2. Register State. In this state, EA starts forming eligible proposer and voter lists. EA can establish a platform or connection to collect proof of eligibility documents from proposers and voters. Proposers must send their blockchain addresses to EA to be able to propose in the election. Voters send their proof of eligibility documents (ID cards, documents, etc.), along with their generated identity commitments.

Each voter generate their identity (ID_i) and identity commitment (IDC_i) with the identity generation function (gen^{ID_i}). It takes a random seed s_i and generates the ID_i which contains a private key and public key pair ($PrivK_i, PubK_i$), a nullifier ($Null_i$) and a trapdoor ($Trap_i$) value as

$$gen^{ID_i}(s_i) \rightarrow ID_i : (PrivK_i, PubK_i, Null_i, Trap_i).$$

ID_i must be kept secret by the voter. The identity commitment IDC_i is constructed by hashing $PubK_i, Null_i$ and $Trap_i$ as

$$h(PubK_i, Null_i, Trap_i) \rightarrow IDC_i.$$

After generating, voter sends IDC_i to EA, so it can be registered as an eligible voter for the election. Note that voter never sends the actual identity ID_i, but a hash of the ID_i which is IDC_i.

EA is also responsible for generating a Merkle-tree root MR_S from all collected identity commitments, i.e., $IDC_{i\epsilon[0..k]}$ where k is the voter count. EA then submits MR_S along with the list of collected IDCs to the smart contract. The smart contract has the main function, *and voters*, for Merkle tree registration. The function can take an arbitrary number of IDCs and a corresponding Merkle root, MR_S. Due to the gas limit of Ethereum, the number of registrable IDCs has an upper bound. In our experiments, we found the upper bound is approximately 30.000 IDCs in a single *addVoters* call. EA can split IDCs into smaller batches (like 30.000 per call) and issue them with multiple calls to the smart contract. Identity commitments (IDCs) must be added to the smart contract in the same order as they are used when constructing the Merkle-tree root. This must be maintained correctly by the EA, so the same Merkle tree can be safely reconstructed by voters. There is a function named *replaceIdCommitments* which replaces the Merkle-tree root and IDCs in the smart contract. This can be used to replace the Merkle tree without destroying the whole contract when EA accidentally registers wrong MR_S. EA also adds proposer blockchain addresses to the smart contract (SC). SC makes sure that the same proposers cannot be registered more than once. EA can manually change the state machine to the next state when it confirms the register state is successfully done. Figure 4.4 shows the sequence diagram for this state.

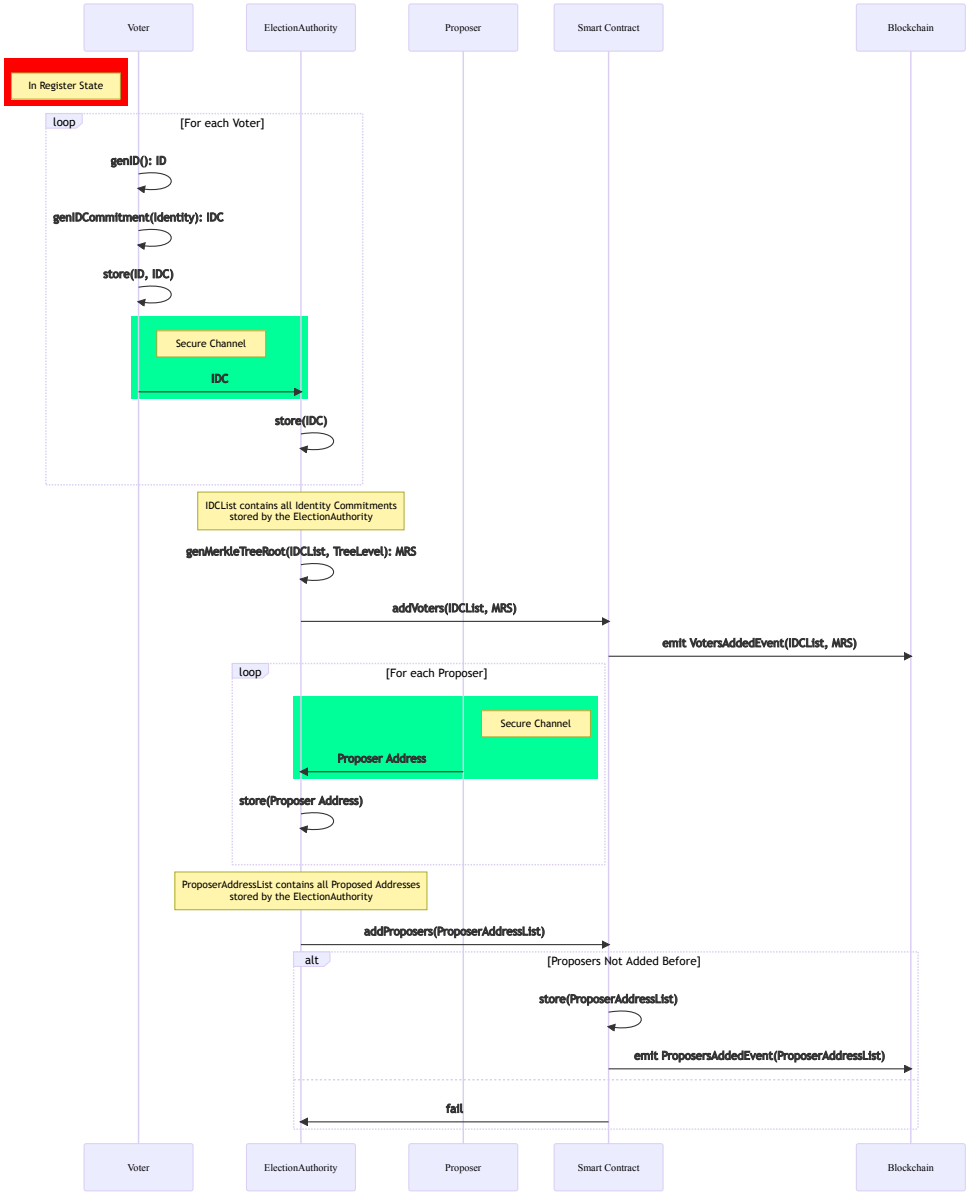

Figure 4.4. Register State Sequence Diagram.

4.2.2.3. Proposal State. At this state, proposers can send their proposals to the smart contract through *propose* calls. Proposals can be considered as the proposed answer strings for a possible solution to the underlying election question. Smart Contract assigns a *candidate ID* (CID_i) for each proposal. Voters use CIDs in their ballots when they cast votes at the *Commit* state. Proposals are not stored in the smart contract, and they are announced and stored in the blockchain as event logs. Each

proposer can propose only once. To ensure this, the smart contract removes proposers from the eligible list after they propose. At the end of the *propose*d call, the smart contract publishes a *ProposedEvent* which contains the assigned candidate ID (CID_i) and the proposal string.

The state has a lifetime *proposal lifetime* which is defined in *Setup* state. The state changes to *Commit* state when this lifetime is exceeded. The state changes when all registered eligible proposers successfully propose. *Proposal* state also changes when the registered proposal count reaches the maximum candidate number to make sure the proposal (candidate) count never exceeds the maximum. Figure 4.5 shows the sequence diagram for this state.

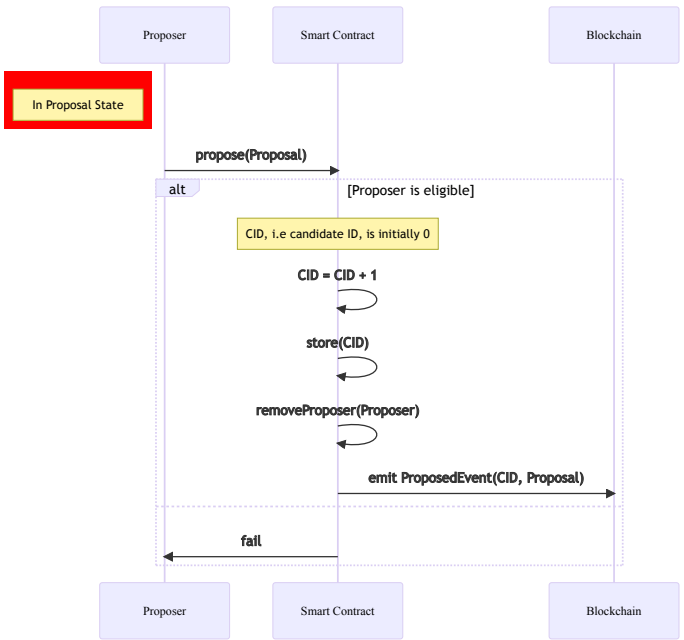

Figure 4.5. Proposal State Sequence Diagram.

4.2.2.4. Commit State. Voters cast their ballots in the *Commit* state. Voters investigate the candidate list by fetching them from the blockchain with filtering events. Recall that in the *Proposal* state, when a proposal is registered, the smart contract emits a *ProposedEvent* which includes candidate ID (CID) and the proposal string. At the *Commit* state, voters can filter these events to gather CIDs and proposal strings, then decide their preference lists accordingly. ElectAnon uses *ranked-choice* electoral system. In this system, voters need to prepare their preferences as a sorted list, from the most preferred to the least preferred. Voters need to prepare a list that contains every registered candidate in the *Proposal* state. It means that the preference list has a static size that is equal to the candidate count (n_c). Each of these preference lists is a permutation of $[1, 2, 3, ..., n_c]$. ElectAnon uses an encoding algorithm to rank/unrank each specific preference list [62]. This algorithm efficiently maps a given permutation list to a single rankID integer, as explained in the Section 4.1.3. Instead of storing the whole list, we store a single voteID (rankID) in the contract. After finalizing their decisions, voters can get the rank of their preference list with the ranking algorithm, which results in voteID (VID_i).

Revealing voteIDs ($VIDs$) during the election violates the *Fairness* requirement. In order to preserve *Fairness*, ElectAnon uses vote hashes VHs. Voters can hash their VID_i with a secret key VSk_i to safely mask their VID_i as

$$h^{keccak256}(VID_i, VSk_i) \rightarrow VH_i.$$

The need for the VSk_i emerges because all available VID_i is publicly known. VIDs are basically in range of $[0, ..., n! - 1]$. Therefore their hash outputs can be calculable with brute-force attacks. In order to mitigate the attack, each voter picks a big random number as VSk_i. This provides a way to generate indistinguishable vote hashes VH_i from a plaintext VID_i. Each voter locally stores their VID_i and VSk_i for later use. They should keep VID_i and VSk_i as secret in order to preserve their privacy.

ElectAnon embeds two functions to provide an easier interface for voters, *getRank* and *unrank*. These convenient functions can assure voters are indeed using the protocol in a correct way, so they can cast their votes safely. These functions are *view* functions. They do not require transactions. Any calls made to these functions are not stored in the blockchain. It means that they can be used without any blockchain address. Voters can use these functions without revealing any particular information about their votes or identities.

Voters need to generate Merkle trees to obtain their Merkle proofs to prove their eligibility to vote. They can fetch Merkle tree leaves, i.e., IDCs, from event logs persisted in the blockchain. Recall that in the *Register* state, when a list of eligible voters is registered, the smart contract emits a *VotersAddedEvent* which includes the registered IDC list and the Merkle root MR_S. Voters also need to obtain Merkle tree level, which is obtainable from the smart contract. With all of these, voters can generate the full Merkle tree and the Merkle proof for the Merkle proof path index (MPI_i) and the Merkle proof path elements (MPE_i) as

$$genTree(IDCList, TreeLevel) \rightarrow MerkleTree$$
$$genMerkleProof(MerkleTree, IDC_i) \rightarrow MerkleProof : (MPI_i, MPE_i).$$

MPI_i contains a list of ones and zeroes which represents directions (left/right) for corresponding Merkle path to the IDC_i. MPE_i contains a list of Merkle nodes in this path. We presented an example Merkle tree in Figure 4.6. In the example tree, IDC list is given as: $[807db9, 9343a4, 5cb255, f4cfad, ba52d21, 643af3, e57c40, e2180c]$. For instance, we can construct MPI_i and MPE_i for the identity commitment $ba52d1$ as follows. As we mentioned, the Merkle path index MPI_i shows directions from root to the Merkle leaf. In this case our Merkle leaf is $ba52d1$, and the corresponding MPI_i is 011 which indicates its position (right,left,left) in the tree. The Merkle proof path elements MPE_i are: $[643af3, 7caf45, 61d143]$. In this example, Merkle root ($d692b1$) can be obtained only with these three elements and the leaf itself ($ba52d21$).

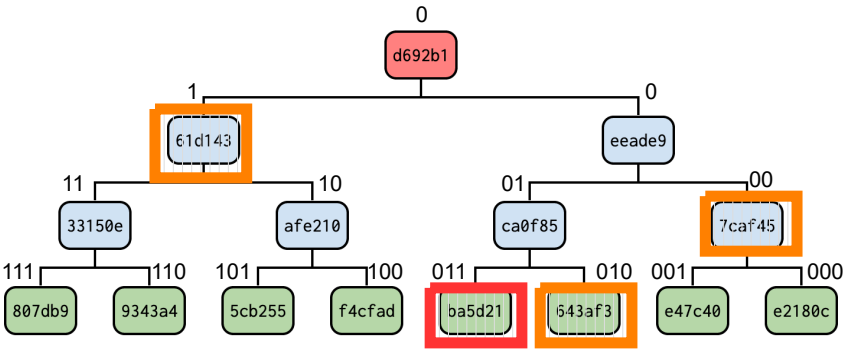

Figure 4.6. Merkle Tree Example.

Upon completion steps above, each voter can start generating its witness and the zero-knowledge proof P_i. Voters use the *Semaphore* circuit to generate their witnesses [56]. Voters need to generate these proofs and witnesses with the same circuit ($Circ$) and the proving key ($ProveK$) used in the *Setup* state so that proofs can be verified correctly by the smart contract. Voters can fetch the circuit ($Circ$) and the proving key ($ProveK$) from the $IPFS$ with the URL provided in the smart contract. The circuit expects following inputs: the vote hash (VH_i), the identity (ID_i), the Merkle proof (MPI_i and MPE_i), the external nullifier ($ExtN$) and a signature ($Sign_i$). The $ExtN$ is defined as the contract address and accessible from the smart contract. $Sign_i$ is the signature on the VH_i which is signed with voters private key $PrivK$. The circuit outputs the witness, which contains a verification for the Merkle root (MR_C) and the nullifier hash NH_i. A detailed explanation about MR_C and NH_i is given at the end of this section. Voters can generate their proofs P_i with witness result along with the $ProveK$ as

$$genWitness(Circ, VH_i, ID_i, MerkleProof, ExtN, Sign_i) \rightarrow Witness : MR_C, NH_i$$

$$genProof(Witness, ProveK) \rightarrow P_i.$$

The *commitVote* function in the smart contract, expects 3 inputs of VH_i, NH_i and P_i from voters. The verifier smart contract (*VerifSC*) expects 2 additional inputs of $ExtN$ and the Merkle tree root MR_S to verify P_i. $ExtN$ is defined as blockchain address of the contract, and the MR_S is the registered Merkle root in *Register* state. The smart contract then proceeds with the verification of given proof P_i to ensure the proof is intact with given inputs. As a result, the contract rejects or accepts the proof. If the proof is accepted, *nullifier hash* NH_i is marked as used to prevent the double-voting. The smart contract stores VH_i keyed with voters blockchain address for later use.

The *Commit* state ends with two conditions. The first one is the timed transition which changes the state after *commitLifetime*. Any transaction (function call) made after this time will be rejected. The other one checks if every voter has committed already. If lifetime exceeds or all voters are done with voting, the smart contract changes to the next state, *Reveal*. Figure 4.7 shows the sequence diagram for this state.

It is worth mentioning how the zero-knowledge proof P_i is constructed and works. In general, the proof ensures these three properties:

(i) The identity is in the eligible member set.
(ii) The same identity is not used to cast a vote hash twice.
(iii) The vote hash is truly generated by the identity which created the proof.

The circuit guarantees the first property with Merkle trees and Merkle proofs. Voter forms the Merkle proof and gives MPI_i and MPE_i as private inputs to the circuit. The circuit also takes each subcomponent of ID_i as private inputs. These subcomponents are $PubK_i$, $Null_i$ and $Trap_i$. The circuit re-generates the IDC_i by hashing $PubK_i$, $Null_i$ and $Trap_i$. The circuit continues with generating the Merkle root MR_C by the generated IDC_i along with given private inputs of MPE_i and MPI_i. The circuit puts the resulting Merkle tree root MR_C into the proof P_i. The smart contract verifies P_i

and checks whether the root in the proof MR_C is verifiable with the registered root MR_S. As a result, the verifier verifies that the voter is able to generate the correct Merkle tree root. With this, ElectAnon verifies that the voter is eligible since the IDC_i is indeed a member of the eligible voter Merkle tree.

The circuit ensures the second property with nullifier hashes. As told before, the circuit takes ID_i subcomponents as private inputs. ID_i contains an *identity nullifier* $Null_i$. The circuit hashes given *identity nullifier* $Null_i$ with *external nullifier* $ExtN$ and the Merkle path index (MPI_i) of IDC_i and obtains NH_i as

$$h(Null_i, ExtN, MPI_i) \rightarrow NH_i.$$

In the verification, the smart contract takes NH_i from the voter as input and verifies that it matches with the one in the P_i. Additionally, the smart contract marks and stores this NH_i as used and invalidates any future calls with the same NH_i. This prevents double voting with the same NH_i. The zero-knowledge proof ensures that NH_i is constructed correctly with components of $Null_i, ExtN, MPI_i$. So reforging a new NH_i would require a change in these components. $ExtN$ is provided by the smart contract itself to the verifier, so the voter has no direct control over it. If the voter provides an invalid $ExtN$ in the proof generation, the verifier will not be able to validate the proof. $Null_i$ and MPI_i is a part of IDC_i, so reforging $Null_i$ would result in a completely different IDC'_i. This would invalidate the proof P_i as this new IDC'_i would not be in the eligible list.

Voters sign their vote hashes (VH_i) with their private key $PrivK_i$. The circuit takes the voter public key $PubK_i$ and the signature $Sign_i$ as private inputs. Then it checks the signature $Sign_i$ with the given public key $PubK_i$. This completes the last property as it can verify that VH_i is indeed generated by the voter.

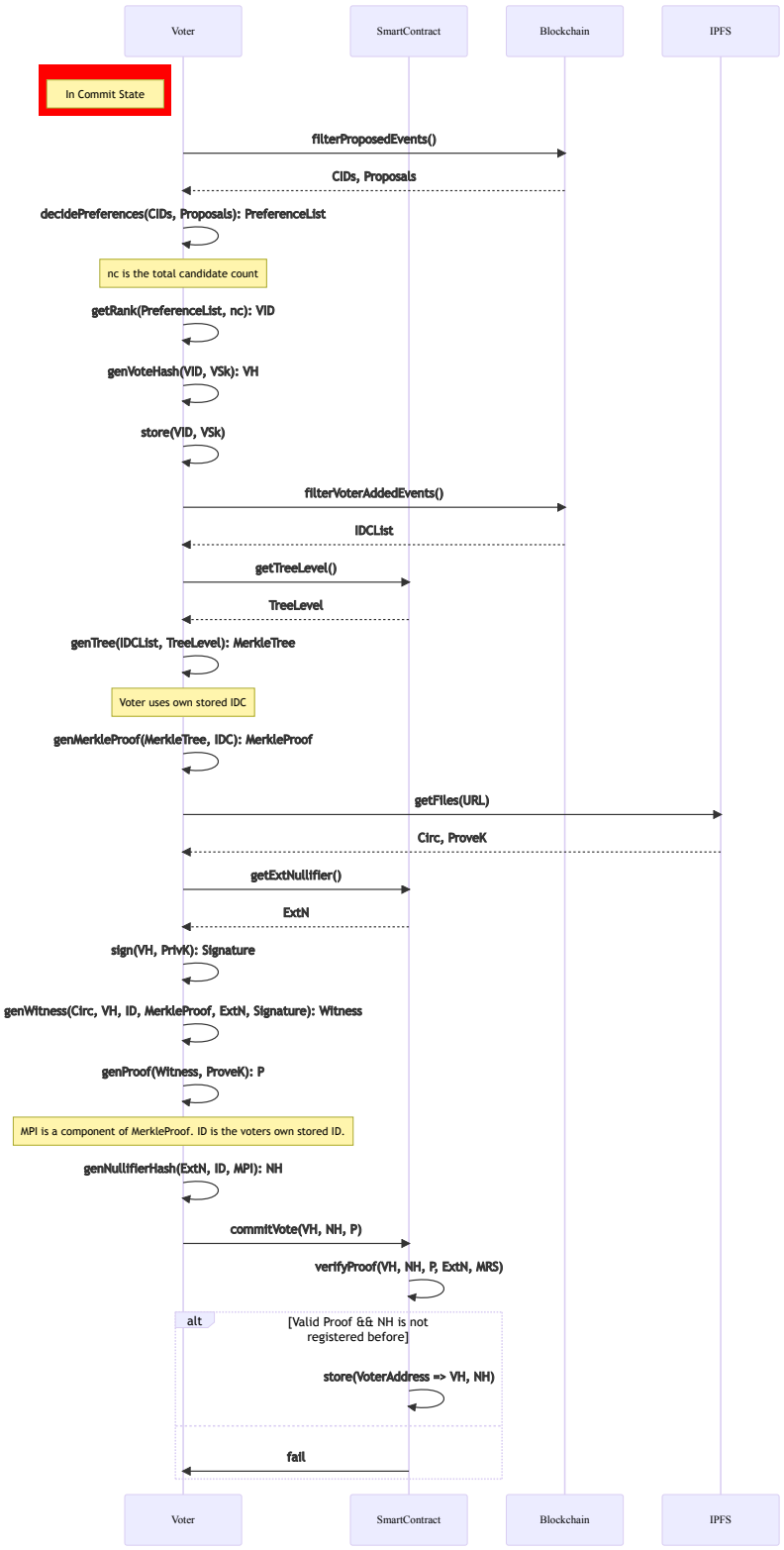

Figure 4.7. Commit State Sequence Diagram.

4.2.2.5. Reveal State. In the *Reveal* state, voters can reveal their vote hashes VH_i with *revealVote* function. The contract stores VH_i in the *Commit* state within a map of *addresses* to *vote hashes*. It means that voters need to reveal their commitments with the same blockchain address they used in *Commit* state. This is to eliminate any attack with brute-forcing *vote secret key* (VSk_i). With this method, an adversary has to seize both (VSk_i), and also the private key of voters blockchain address that is used in *Commit* state to reveal votes. Each voter provides inputs of *voteID* (VID_i) and vote secret key VSk_i to the smart contract. The smart contract checks if the hash of these two inputs (VH'_i) is equal to the one that was stored (VH_i) in the previous *Commit* state as

$$h^{keccak256}(VID_i, VSk_i) \rightarrow VH'_i$$
$$VH'_i \stackrel{?}{=} VH_i.$$

If they're not equal, the smart contract rejects the transaction. Otherwise, it deletes the stored VH_i in the contract to guarantee that it is not revealed twice.

The contract passes VID_i, candidate count n_c and a storage mapping *Tally State* TS to the tally library. TS is required to keep revealed votes in the storage so that tally libraries can use revealed votes. The tally library defines a *tally* function, which tallies revealed results. The library is capable of changing the election state with TS, which means that each revealed VID is counted accordingly and put into the contract storage through TS. The *tally* function may differ for each tallying algorithm and implementation. We implemented two tallying libraries for *Borda Count* and *Tideman* in our work. The details are discussed in Section 4.3.1.

After all committed votes are successfully revealed, this state ends, and the smart contract changes to *Completed* state. There is also a timed-transition *revealLifetime* for this state. Voters must reveal their votes within this lifetime; otherwise, their votes cannot be counted. This is the last state where actors can issue state-changing transactions for the ElectAnon. Figure 4.8 shows the sequence diagram for this state.

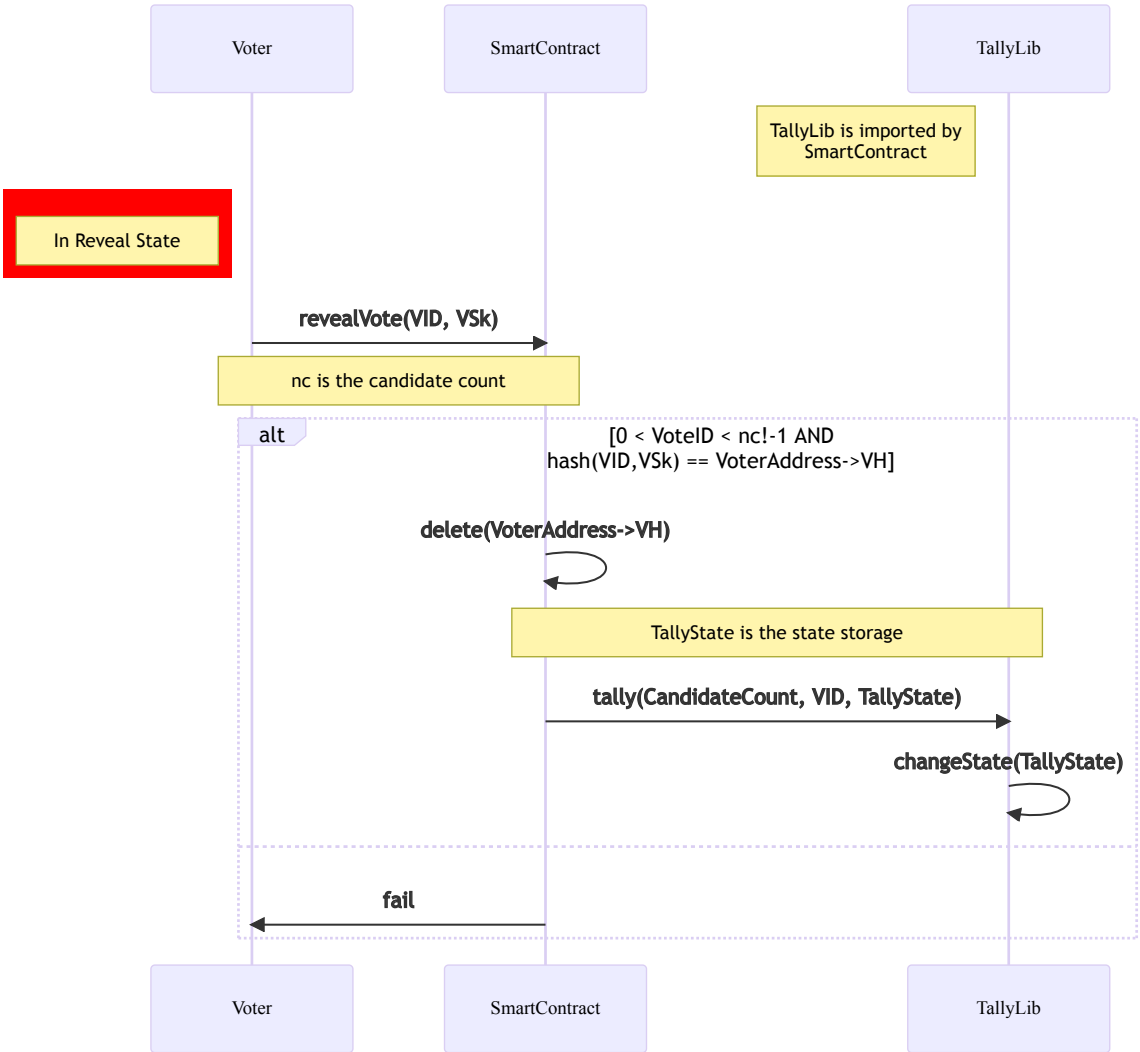

Figure 4.8. Reveal State Sequence Diagram.

4.2.2.6. Completed State. The *Completed* state is not an actual timed-state. This is because if block number exceeds the *revealLifetime*, any call to the *revealVote* function is immediately rejected. It means that no state change occurs on the contract, so actually, the state cannot be changed from *Reveal* to *Completed* in that case. This is because smart contracts cannot change any state without an actual user trigger; this is a general EVM smart contract limitation. This does not change anything in the voting protocol as *Reveal* state is not callable after *reveal lifetime*. If all committed votes are revealed, the smart contract changes the state to *Completed*. In this case, whoever reveals the last commit will trigger a state change from *Reveal* to *Completed*.

In this state everyone can call the *electionResult* function to get the election result. This function is a *view* function, meaning that calling it will not change the state in the blockchain. Thus it requires neither transactions nor fees. The smart contract uses an inner tallying library to calculate the results. The tallying library fetches the tallying state TS, the same storage that is populated in the *Reveal* state. The tallying library has *calculateResult* which takes the candidate to count n_c and tally storage TS then interprets the election result. The election result shows the candidate ID (CID_i) of the winner. Note that, the smart contract publishes *ProposedEvent* for each proposal in the *Proposal* state. The actual proposal string can be fetched by filtering *ProposedEvent* with the winner CID_i to find the winner proposal. The election result can be calculable with different tallying libraries. Figure 4.9 shows the sequence diagram for this state.

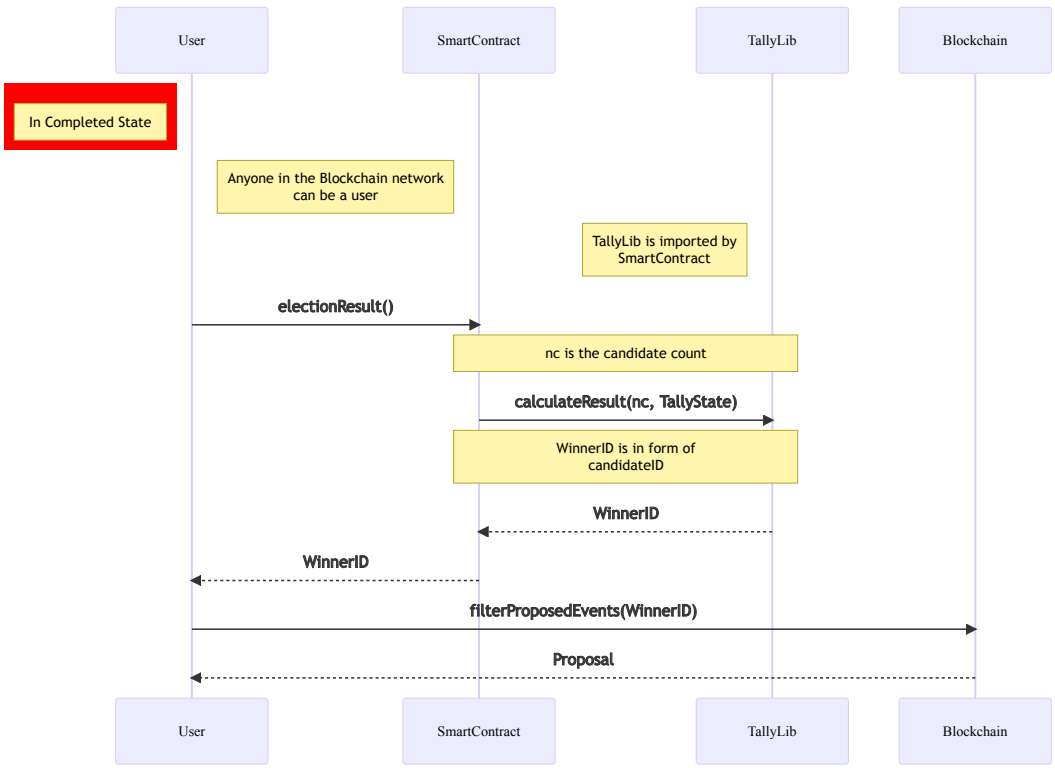

Figure 4.9. Completed State Sequence Diagram.

4.3. Technical Details

In this section, we provide some technical details regarding technical preferences like algorithms and optimizations. A full-fledged class diagram is presented under the Appendix in Figure A.1.

4.3.1. Tallying Libraries

Solidity libraries can be used as shared codes for smart contracts. Smart contracts can load functions and types from these libraries. Our tallying algorithms are implemented with libraries. Hence it's trivial to change the tallying algorithm by loading a different library before the deployment. The main smart contract can load another tallying library by importing it. This change should be done before the contract deployment. Ethereum does not allow any change in the contract code after the deployment. A different tallying algorithm can be used through libraries while still keeping the core functionalities of the main smart contract like *revealVote* and *commitVote*. Tallying libraries has two main functions, *tally* and *electionResult*. The *tally* is used in *Reveal* state and is responsible for counting the votes and putting them into the tallying storage TS. The *electionResult* function is used in *Completed* state and announces the winner ID by interpreting the given tallying storage TS. These two functions can differ for each tallying algorithm.

4.3.1.1. Borda Count Library. The history of Borda Count method roots back to 15^{th} century [65]. In the election of the Holy Roman Emperor, Nicholas of Cusa proposed to score each candidate by putting a number ranging from one to the total number of candidates. However, the Borda Count was named after Jean Charles de Borda, who devised and scientifically analyzed the method in 1781 [65]. In the basic form, each ballot holds a sorted list of candidates; each of these candidates gets a score based on their orders in the list. For example in a five-candidate election, where each candidate has an ID between $c_1, c_2, ...c_5$, a sample ballot would be $[c_3, c_2, c_1, c_4, c_5]$. This is equivalent to giving c_3 the maximum score of 5, c_2 score of 4 and so on. At the end of

the election, each of these scores for the candidateID is summed together. As a result, the candidate with the maximum total score wins the election.

We implement Borda Count tallying algorithm as a tallying library. The *tally* function takes voteID (VID) and then unranks it into the related preference list. The preference list represents a sorted list of candidates. The function scores each of sorted candidates in a decreasing scores of $[n_c, n_c - 1, n_c - 2, ..., 1]$, where n_c is the candidate count, i.e preference list size. These scores are added to the tally state storage TS in a map of candidateIDs to their respective cumulative score. The *calculateResult* function of this library compares each cumulative score of candidates given in TS, then returns the candidateID with the maximum score. We used this library in our experiments in the Section 5.2.

4.3.1.2. Tideman Library. The Tideman Method [63], also known as *Ranked pairs*, is a ranked-choice-based tallying algorithm. The method collects the ranked preferences and compares each candidate in a pairwise fashion. Then each of these pairwise comparisons is sorted by their winner's vote dominance against the loser. The algorithm starts locking the winners against losers in this sorted order by constructing a directed graph. The one that does not being locked by another candidate, i.e., the vertex that has an in-degree of 0, becomes the winner. If any cycle occurs in the locking, that pair is ignored. The method is a *Condorcet* method as it guarantees the winner wins every head-to-head match against other candidates. The author [63] states that *Borda* is not a Condorcet method unlike the Tideman method. *Majority rule* ensures that a candidate is the election winner if it is selected as the first choice by the majority. The Tideman guarantees the Majority Rule [63]. Unlike Tideman, Borda does not guarantee the Majority Rule [66].

We implement the Tideman Method as a tallying library. The *tally* function increments the count seen voteIDs (VID) and then stores the count in the tally state TS. Unlike *borda count* library, *tally* does not unrank VID into the preference list. The *calculateResult* function unranks VIDs into preference lists first. Then it follows

the Tideman algorithm to calculate the winner. The algorithm heavily uses graphs, matrixes, and sorting algorithms. Due to the nature of smart contracts, these operations are very costly. We were able to reduce some of these costs by efficiently counting and storing the preferences via the rank/unranking permutation algorithm in the *tally* function. Without such an encoding scheme, storing preferences as lists would make it even costlier. *calculateResult* function, is a *view* function. So it does not actually cost any transaction fee, so we can say it's free to run. However, the block limit is still applicable to the *view* functions even they're practically free. Ethereum nodes do not return results in case of this block gas limit is reached by a function call. So there is still a gas limit applied to the *view* functions. We've seen that our Tideman algorithm is capable of running with 250 voters and ten candidates. This makes it a viable option for small-scale elections, but it's not feasible to be utilized in a large-scale election. The Tideman gas results can be found in Table 4.2.

Table 4.2. Tideman Gas Results (voterCount=250, candidateCount=10).

Entity: Transaction	Gas Cost
Deployment	4,065,760
addVoters	276,929
addProposers	286,064
propose	42,155
toProposalState	71,877
commitVote	312,382
revealVote	84,454
electionResult	21,113,398

4.3.2. Block Number

We use timed transition for our state machine. It means that our smart contract must have a perception of time. Ethereum Virtual Machine (EVM) provides a way to fetch timestamps of blocks. The *block.timestamp* can be used in smart contracts

to fetch the timestamp of the current block. However, there is a known vulnerability with this method [67]. The block timestamp can be alterable by the block miner up to some margin, approximately 30 seconds. This actually does not possess a very high risk for voting scenarios since election phases have relatively large lifetimes that can be measured in hours or even days. However, we have addressed this issue by using *block.number* instead of *block.timestamp*. *block.number* is not alterable by the miner and still can be used as a source of time. There are approximately 15 seconds between each block. The network adjusts mining difficulty to stay in this interval. We specified state lifetimes in terms of block numbers.

4.3.3. Batch Inputs

Most of the blockchain platforms require a base transaction fee. Ethereum requires an additional 21.000 gas for each transaction. This fee is included in every transaction. It means that every single transaction costs additional fees. We tried to minimize fee requirements and gas consumption by using batched inputs whenever possible. Functions that use batched inputs can reduce the transaction count, so it can reduce the total required transaction fees. For example, instead of adding a single eligible proposer address, we can batch multiple addresses into a single transaction. In this way, we can reduce the total transaction count that needs to be done for adding eligible addresses. There are mainly two functions that effectively benefit from batched inputs. *addProposers* can be used with multiple proposer address inputs. This is a viable option in the case that the election authority collects the eligible proposer list beforehand. So the election authority can issue a single transaction that contains all eligible proposer addresses. *addVoters* also uses batched inputs by adding multiple voters to the eligible voter list in a single transaction. This also reduces the transaction count, thus reducing transaction fees.

4.3.4. Storage Costs

ElectAnon also aims to reduce storage costs by omitting any unnecessary storage operations. We offloaded some storage costs to *events*. Smart contracts can publish persisted logs to the blockchain through events. Applications can subscribe to certain event topics and get notified about these published events. One of the use cases of events is to reduce the smart contract storage by offloading them to events. Events consume much less gas than storage in certain cases, especially when it comes to store complex types like *arrays* and *strings* [68]. For instance, we did not store any proposal string in the contract. The smart contract publishes an event with the proposed string when a proposal is added. So voters can listen *Proposed* event topic and fetch these proposals and decide their preferences based on strings in the proposal events. Moreover, when voter identity commitments are added to the smart contract, an event is published instead of storing these in the contract. The contract itself does not need to know all of these commitments; it just needs to hold the Merkle tree root. So ElectAnon only stores Merkle tree root in the contract. Voters can subscribe to *VoterIdCommitsAdded* topic and store ID commitments (IDC_i) so that they can reconstruct the tree and find their own commitment Merkle paths. In our implementation, we tried to reduce the number of state-changing functions by using *view* functions wherever possible. For instance, the self-tallying mechanism *electionResult* function is a *view* function. This enables users to interact with the function and fetch the election result without paying any transaction fee. ElectAnon also does not store election results as they can be queried with a gas-free view function *electionResult*.

4.3.5. Optimized Merkle Tree

ElectAnon slightly improves smart contracts of *Semaphore*. The smart contract in Semaphore inserts a single leaf (identity commitment) to the Merkle tree at a time [56]. It also generates the Merkle tree root on the smart contract. The single leaf insertion requires a traversal and update on the tree. The tree is reformed by hashing each level of the corresponding subtree with the new leaf. *Semaphore* uses a hash function called

MiMC. *MiMC* provides an efficient hash solution for SNARKs by reducing required multiplicative complexity [69]. Unfortunately the *MiMC* is not an optimized hash function for the EVM, a single hash operation requires almost 30.000 gas [70]. A single leaf insertion to a 20-level tree consumes almost 500.000 gas in our experiments. At the time of writing this work, the value of 500.000 gas is approximately 200 USD. This is not scalable for large-scale elections. We optimized this function by inserting multiple leaves with a single transaction. In our work, we assumed that identity commitments could be known by the election authority beforehand. So with this assumption, we offloaded Merkle tree calculations to off-chain where election authorities calculate the Merkle tree roots in their local environments. The optimized version takes multiple leaves (identity commitments) and a single Merkle tree root. With these inputs, Merkle tree calculations need not be done in the smart contract, so they don't consume gas in large quantities. This approach reduces not only the gas consumption of a single insertion function but also the number of calls to this function as we insert multiple leaves at a time. Our optimized version costs nearly 560.000 gas to insert 750 leaves at a time. This is almost equal to the Semaphore's single leaf insertion gas cost.

4.4. Extensions

In this section, we discuss some possible extensions of our work. They mostly depend on specific use cases and setups.

4.4.1. Multiple Elections

It's possible to conduct several elections with the same Merkle tree, i.e., eligible voters list. There are essentially two smart contracts; one of them includes the optimized Semaphore contract, which contains the zero-knowledge verifier and the Merkle tree registration. The other smart contract contains voting functions like ballot casting, revealing, and tallying. Currently, ElectAnon is optimized to run a single election. ElectAnon merges these two contracts into one and creates a single deployable smart contract to host a single election. However, it is possible to separate these smart con-

tracts and deploy them as two different instances. In this case, the deployed Semaphore contract will have its own contract address. The Merkle tree, thus eligible users, can be registered directly to this Semaphore contract instance. Afterward, other election contracts, like ElectAnon, can use this Merkle tree through delegated calls. Remind that *external nullifier* $ExtN$ is used in *nullifier hash* NH_i to prevent double-signalling, i.e double-voting. In this case, every election contract can register itself to the Semaphore contract with a different $ExtN$. With this method, other election contracts can still ensure that no vote is being double signaled within the same election. However, it also means that a voter can still vote for different elections at the same time. This would effectively help to scale repeated elections or multiple elections since one single Merkle tree can be used in multiple elections. However, at the same time, it would increase the cost for a single election as *delegateCalls* requires extra gas cost. Also, deploying two contracts instead of one increases the total gas cost further.

4.4.2. Assisted Merkle Tree

The Merkle tree of identity commitments (IDC_i) must be known by the network so that each voter can generate their Merkle proofs and paths for the zero-knowledge circuit. ElectAnon takes identity commitments as inputs along with the Merkle tree root in the *addVoters* function. These identity commitments are published with Ethereum events. Voters can subscribe to these events and fetch Merkle tree leaves and the root. This approach ensures that the Merkle leaves (identity commitments) are available to voters via blockchain. However, it also means that the gas consumption of *addVoters* increases with the voter count since the function takes a list of *identity commitments* as input. The smart contract actually does not require any of these leaves; it only needs to know the Merkle tree root for proof verification. So the identity commitments list input can be omitted from the smart contract function. This would require *election authority* to publish identity commitments in an external, public channel; so that voters still can generate their Merkle proofs correctly. The method adds an additional trust assumption on the election authority, as the network depends on the authority to publish identity commitments.

4.4.3. Merkle Forest

ElectAnon uses Merkle trees in order to efficiently prove the eligibility of voter identity commitments. On the other hand, the original Semaphore smart contract calculates the Merkle tree on the smart contract (on-chain) as mentioned in Section 4.3.5. Calculating the Merkle tree on-chain provides a safe way to keep the Merkle tree root intact with the registered leaves. This calculation has a very high gas consumption since it requires multiple hash operations per tree level. We removed this by offloading the calculation to the election authority. The Election Authority calculates the Merkle tree root off-chain then sends it to the smart contract, along with the leaves (identity commitments) of the Merkle tree in the *Register* state. However, this puts trust in the authority as we trust the authority to calculate it honestly. We can reduce this trust possibly with two approaches: an on-chain challenge and verifiable local computation.

It's almost impossible to completely eliminate the trust in the authority since the authority has to form an eligible list. So the authority can always forge an invalid list by not putting some voters on the list; or putting non-existent voters into the list. We assume that the election authority has no interest in forging an invalid eligible list. In this section, we discuss the trust in the authority to form a valid Merkle tree.

The first solution would be to calculate the Merkle tree on chain but only when it's needed. The approach includes a challenger and a verifier. Election authority still computes the Merkle tree locally and sends it to the smart contract along with the whole Merkle tree. A challenger can challenge the registered Merkle root with a Merkle path in a pre-defined time frame. Merkle path includes a direction path of where the verification should start from. Then smart contract can try to regenerate the Merkle tree root with the given path and compare it with the registered root. Since it would require a gas consumption for the challenger to issue a challenge transaction for the smart contract, there can be some incentive mechanisms to compensate for this challenge cost. The smart contract can reward the challenger if the registered Merkle tree root is indeed invalid. In a successful challenge case, where Merkle tree

root is found invalid, the smart contract can halt the process and require a valid Merkle tree to be registered again. This approach is similar to what the original semaphore contract does, but instead of recalculating the Merkle tree in every leaf register, it can be calculated when it's challenged. However, the solution requires Merkle tree hashes to be stored on the smart contract, which increases the gas cost directly proportional to the leaf count. Generating the Merkle tree also requires too much gas since it involves *MiMC* hash function, which is not natively supported by EVM. The solution depends on a challenger who has either a good incentive or motivation to challenge the registered tree. It means that the security of this solution depends on the motivation of challengers.

We propose *Merkle Forests* solution, which includes zero-knowledge proofs. In this solution, each Merkle tree root can be calculated with a zero-knowledge circuit, and then calculation proofs can be verified on the chain. The circuit takes a fixed-size leaves list as a private input. Then it forms a Merkle tree from these leaves by using *MiMC* hash function and generates the root. It also hashes elements in the given list input with *keccak256* hash function. The proof outputs both the Merkle tree root and the hash of inputs. The verifier smart contract takes the leaf-list, the Merkle tree root, and the zero-knowledge proof as inputs. Firstly, the verifier smart contract computes *keccak256* hash of the given leaf-list. Then it verifies the given zero-knowledge proof with the given Merkle tree root and the calculated hash of inputs. In this way, the smart contract can ensure that the given Merkle tree root is indeed calculated with the given leaf-list. After a successful verification, the root is registered to the ElectAnon smart contract with a tree index. This is because zk-SNARK circuits do not support dynamic-size arrays. So in order to register more voters, a new tree must be constructed and registered to the smart contract. The number of trees need to be registered is directly proportional to the voter count. Let's say tree leaf size is set to 256, so each tree can hold 256 voters. It means that if there are 2560 eligible voters in the election, ten trees must be registered to the smart contract. Smart contract stores tree roots within a *mapping[treeIndex] => treeRoot* structure. The smart contract publishes an event when a tree is registered. The event contains *treeIndex, treeRoot*

and *leaf-list*, so that voters can track which tree they're registered in. In the *commit* state, each voter must explicitly tell which tree they are into the smart contract; so that the smart contract fetches the Merkle root of that particular tree and verify the Semaphore proof of the voter. Note that ElectAnon already has a zero-knowledge proof construction from *Semaphore*. This Merkle Forest approach adds another zero-knowledge circuit and verifier to the ElectAnon to verify that Merkle tree constructions are done faithfully. With this method, the smart contract can verify that each Merkle tree root input is correctly constructed from the given Merkle tree leaves.

We have realized that *keccak256* hash function is not very efficient for zero-knowledge proofs. However, it is an efficient hash function for EVM-based smart contracts. We have also realized that we can accomplish the proposed *Merkle Forest* approach by not using keccak functions at all. So we wanted to compare these two different versions. Keccak hash function is only required when leaf inputs are private and not verifiable by contract. However, if we can remove this necessity by making the leaf-list as public input, we can also remove the keccak hash function from the circuit. With that, the verifier smart contract can take the full leaf-list as the input and verify the remaining Merkle tree construction proof with this list input. In this work, we also implemented a circuit that can take *public* inputs of tree leaves and requires no keccak hash function. We used *Zokrates* [55] as our main zk-SNARK toolbox in our implementations. We conducted experiments for both of these Merkle Forest implementations (with and without keccak256) in the Section 5.2.1.

5. ANALYSIS

5.1. Security Analysis

In this section we analyze ElectAnon with aspects defined in the *Election requirement* section.

5.1.1. Eligibility

At the beginning of the *Register* state, the election authority (EA) requests verification documents from voters. Each voter generates an identity commitment IDC_i and submits it to EA in a secure channel. EA decides voters' eligibility by verifying their provided verification documents. After deciding eligible voters, EA forms a Merkle-tree with voters identity commitments (IDC_1, IDC_2,...IDC_k). EA registers the Merkle tree root (MR_S) and identity commitments (IDC_i) to the smart contract at the end of the *Register* state. In the *Commit* state, the protocol verifies that each voter owns a valid identity commitment IDC_i with zero-knowledge proofs created by the *Semaphore* circuit. The circuit also takes the Merkle proof of IDC_i, which consists of Merkle proof path indexes (MPI_i) and Merkle proof path elements (MPE_i). Then the circuit generates the Merkle tree root (MR_C) with the given inputs puts it into the proof P_i. The verifier smart contract verifies that the MR_C in the proof and the registered Merkle tree root in the smart contract MR_S matches with each other. At the end of this verification, the protocol verifies that the IDC_i is indeed in the eligible voter set. The smart contract rejects transactions without valid proof; thus, ElectAnon ensures *Eligibility*.

5.1.2. Uniqueness

Uniqueness is ensured by *Semaphore*'s double-signalling prevention. In the *Setup* state, each voter generates their identity ID_i and identity commitments IDC_i. This

ID_i, and therefore IDC_i, includes a random-secret nullifier $Null_i$. Each voter generates the zero-knowledge proof P_i with private inputs of their nullifier $Null_i$ and an external nullifier $ExtN$. The circuit hashes these private inputs and generates NH_i as follows $NH_i = h(Null_i, ExtN, MPI_i)$. Smart contract verifies the proof P_i to make sure NH_i is correctly generated by the voter. Smart contract also stores this NH_i and invalidates any call with the same NH_i. Each of these NH_is are unique to their ID_i, it means that in order to reforge a valid NH_i adversaries must register a new eligible ID_i, which contradicts with the *Eligibility* proof. Since no voter can cast more than one vote in the election, ElectAnon achieves *Uniqueness*.

5.1.3. Privacy and Anonymity

Privacy is ensured by preserving the anonymity of voter identities. We assume voters send their identity commitments (IDC_i) to *Election Authority* (EA) in a secure channel. EA can also collect any documents or information from voters to decide their eligibility. The protocol aims to keep identities ID_i and their commitments IDC_i secret when voters cast their votes. This is because if one of them is revealed in the voting time, then EA can distinguish voters and learn about their votes. Most of the existing works encrypt/hide the actual votes, but they do not consider hiding the voter identities. ElectAnon follows a contrary way to preserve privacy. In ElectAnon, we aim to hide voter identities as well.

In the *Commit* state, voters generate zero-knowledge proofs without revealing their identity commitments ID_i or IDC_i. The zero-knowledge proof P_i proves that the vote is indeed generated by the voter. Voters generate their zero-knowledge proofs in their local offline environments. The zero-knowledge circuit $Circ$ takes voter identity ID_i as a private input and generates the proof P_i. By taking a private input, $Circ$ guarantees that given ID_i is not revealed in the proof. The smart contract takes P_i as input but does not require ID_i or IDC_i. As a result, no identity ID_i or their commitment counterpart IDC_i is revealed.

Voters still interact with the smart contract through their blockchain addresses. We assume they use a new and fresh blockchain address when they first interact with the smart contract in *Commit* state. Any reuse of these addresses in other blockchain applications can compromise their anonymity. This is not a specific cause for our protocol but a common situation for blockchain applications. If both of these assumptions hold, then even the EA cannot deduct any information about voter identities since IDC_i is not revealed in the protocol. In the *Reveal* state, voters expose their one-time-only vote secret keys VSk_i which are not related to their identities. In that phase, voters use the same blockchain addresses that they used in the previous *Commit* state. However, this will not have any impact on the anonymity since they do not use any information about their ID_i or IDC_i in the *Reveal* state. ElectAnon ensures the *Anonymity* since no one can distinguish any voter operation and link them to actual voter identities. This satisfies *Privacy* since no one also can link a vote to the actual voter identity.

5.1.4. Fairness

ElectAnon voting phase consists of two consecutive states, *Commit* and *Reveal*. Recall that voters commit the hash of the vote (VH_i) without revealing the actual vote ID (VID_i) in the *Commit* state. This is done by hashing the VID_i with a random vote key (VSk_i), e.g $VH_i = h(VID_i, VSk_i)$. These VSk_i and the VID_i must be kept secret until end of the *Commit* state to guarantee *Fairness* is preserved. In the *Reveal* state, voters provide their plaintext VID_i along with VSk_i. The smart contract verifies the stored VH_i can be constructed with these inputs by checking $m[senderAddress] : VH_i == keccak256(VID_i, VSk_i)$. Then these revealed votes are stored in the smart contract to be tallied. This two-step voting phase ensures that committed vote hashes do not reveal any information about actual votes and committed votes are not modifiable after the commitment. *Fairness* defined as "no intermediate results should be available to be obtained." ElecAnon guarantees that no-intermediate results are obtainable in the *Commit* state, and committed votes are not alterable. Thus, ElectAnon satisfies the *Fairness*.

5.1.5. Soundness

ElectAnon encodes ranked-choice lists into single integers, i.e to vote IDs $VIDs$. Each of these VIDs represents a permutation of candidate IDs. In the *Commmit* state, voters commit their preferred VID_i with hashing them into vote hashes VH_i. Solidity smart contracts can encode data with their types in hash functions by using *The Contract Application Binary Interface (ABI)* [71]. We use *ABI* to make sure hashed commitments also carry their type information so that we eliminate any type-related corruption in the data. In the *Reveal* state, voters reveal their VID_i. Each of VID_i integers represents an encoded ranked-choice list. ElectAnon decodes these VID_i with the unrank algorithm [62] to obtain actual ranked-choice preference lists. For a list of size n_c, there can be at most $n_c!$ permutations. It means that the valid VIDs can be in range of $[0, n_c! - 1]$. This is easily detectable in the *Reveal* state as VIDs are revealed in this state. ElectAnon checks the revealed VID_i and rejects the transaction if it is not in the valid range. *Soundness* is defined as "no invalid ballots should be tallied." ElectAnon achieves *Soundness* by securing that no invalid votes can be revealed and tallied.

We explicitly reject invalid votes instead of marking and storing them in the smart contract. This reduces extra storage costs. In the *Commit* state, the smart contract cannot perform a sanity check on votes (VID) since they are not committed in plaintext but with their hashes (VH). It means that invalid votes can be committed in the *Commit* state. However, in the *Reveal* state, we prevent invalid votes from being revealed and tallied by rejecting transactions that contain invalid votes. We provide log feedback stating that the registered vote is invalid when voters try to reveal an invalid vote. Moreover, there are two *getRank* and *unrank* functions that can be used to encode/decode preference lists into VIDs.

5.1.6. Universal Verifiability

ElectAnon is a self-tallying protocol, and it uses blockchain technology. The technology ensures that every transaction is transparent and verifiable by the network. With these two aspects, anyone can verify each commitment or reveal transactions. Tallied results also can be verifiable with revealed voteIDs ($VIDs$) at the end of the reveal phase.

5.1.7. Individual Verifiability

Voters can verify that their committed votes are successfully cast on the ballot by verifying blockchain transactions. This can be done by verifying that transactions contain their committed vote hashes (VHs). They can also make sure that their revealed votes are tallied correctly by verifying that transactions contain their plaintext (VID) input in the *Reveal* state transaction.

5.1.8. Robustness

We define a state-machine in the smart contract to ensure each state transition is well executed and election protocol flows without any interruption. This certainly makes it possible to conduct an election without a manual transition from authority. The authority is only available in *Setup* and *Register* states. The smart contract ensures that the Election Authority (EA) cannot call any state-changing function after *Register* state. EA can only disrupt the election before voting starts, i.e., before the *Commit* state. There is no voter-smart contract interaction required before the *Commit* state. So if EA disrupts the election before the *Commit* begins, it would only harm the EA itself.

The smart contract (SC), the circuit ($Circ$), and zero-knowledge keys ($VerifyK$, $ProveK$) are announced publicly, and thus everyone in the network can verify thems. EA generates the zero-knowledge verifier smart contract using the verification key.

Note that, zero-knowledge key setup derives $VerifyK$ and $ProveK$ from a set of public parameters ($Params$). These public parameters are also announced publicly at the end of the *Perpetual Power of Tau* [59] ceremony. It means that even if EA deploys an invalid verifier contract, it is immediately detectable. Voters also will not be able to use this invalid smart contract with their valid proving keys. This situation would be to the disadvantage of the election authority (EA) since no voter can interact with the invalid in the smart contract; thus, the election can never start. In the voting process, every voter verifies their own proof.

We do not use any shared-key encryption to ensure voters' activities do not affect each other. So our protocol is safe against *abandoning vote* attacks, which can be seen in works like McCorry et al. [41], Yang et al. [35] and Priscore [37]. In the *Reveal* state, voters reveal their own votes. If a voter abandons a committed vote, others will not be affected by it, and the election can safely continue to tally.

ElectAnon guarantees that a started election will not be disrupted by any means. As a result ElectAnon achieves *voting availability*. Moreover, ElectAnon is a self-tallying protocol. It means that everyone can tally the result without requiring any external assistance. As a result, ElectAnon also achieves the *tallying availability*. ElectAnon guarantees *Robustness* with these two achievements combined.

5.2. Experiments and Results

We deployed ElectAnon to a local Ethereum network by using Hardhat [72] tool. Hardhat is a development environment that provides local Ethereum networks, gas consumption reports, a high-level language to conduct tests, and a wallet pre-filled with accounts. Hardhat also supports smart contract language *Solidity*. We wrote tests in NodeJS and used a modified version of Semaphore library *libsemaphore* [57] to generate proofs and witnesses. We used Solidity version 0.8.7 and the latest Ethereum fork *London*. We used a MacBook Pro with an 8-core 3.2GHz Apple M1 chip and 16 GB Ram, running on macOS BigSur (Version 11) to conduct our tests.

Figure 5.1 shows measured gas consumption results for *commitVote*, *revealVote* and *electionResult* functions. These functions are mainly called by voters in the election. We run them in 2 different setups. In the first one, we fixed the voterCount to 10 and increased the candidate count linearly. In the second setup, we used a fixed candidate count of 10 and increased voter counts exponentially. We found that *commitVote* has a $O(1)$ gas complexity since it is not affected by the candidate count change or vote count change. The function consumes approximately 315.000 gas per transaction. The function verifies the zero-knowledge proofs. Hence, it consumes relatively more gas than other functions. We have seen that the function *revealVote* is only affected by the candidate count. The consumption function is approximately: $n_c * 8000 + 39000$ and thus linear with $O(n_c)$. The votes are counted in *revealVote* function and voteIDs (VID) are unranked with the Ranking/Unranking permutation algorithm [62]. The algorithm takes a linear time to unrank the rankID to a permutation list. As expected, it causes a linear gas consumption with respect to candidate count n_c.

We found that voter count has no effect on *revealVote*. There is a down-slope in between voterCount=10 and 100. This is due to the fact that there is an additional map initialization cost when *revealVote* is called for the first time. Hardhat gas-reporter takes an average of gas consumption if the same function is used more than once. Eventually, the average value becomes closer to the maximum when there are few voters and becomes closer to the minimum when there are more voters.

The *electionResult* function is not affected by the vote count but is linearly affected by the candidate count $O(n_c)$. This is because the *Borda Count* tally library partially tallies the results in the *revealVote* function, as previously mentioned in the Subsection 4.3.1.1. The *electionResult* function iterates over each candidate ID in the tally storage TS and finds the candidate with the maximum score, then reports it. As a result, the function cost increases linearly with respect to candidate count.

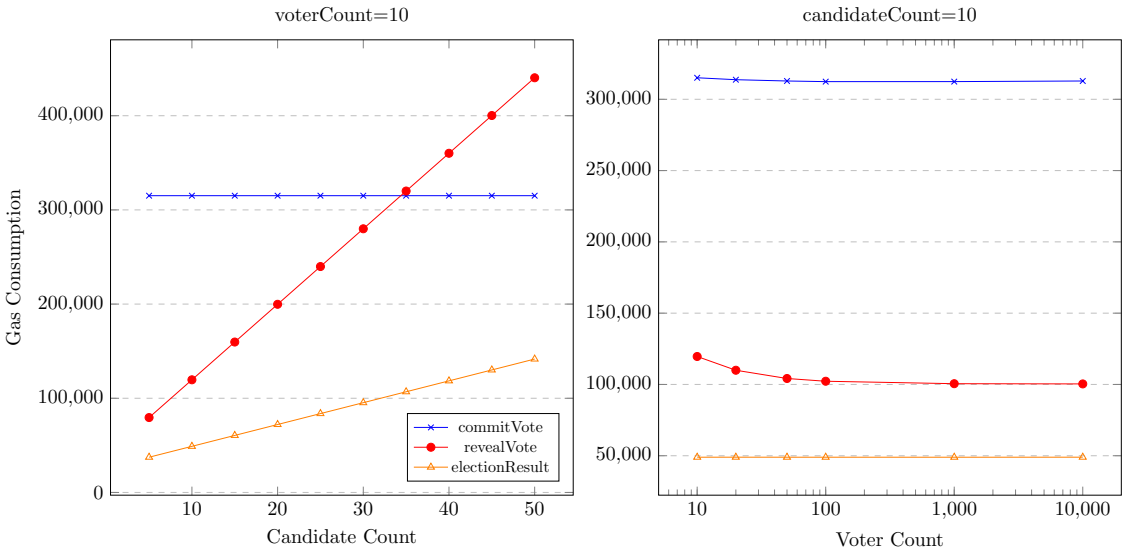

Figure 5.1. Voter Functions.

The smart contract deployment takes a total of 3458406 gas. The deployment is done only once by the election authority. Figure 5.2 shows gas consumptions for *addVoters*, *addProposers*, *propose*, *toProposalState* which are mainly used by election authority and proposers. The gas consumption of *addProposers* is proportional to the candidate (proposal) count. The cost function of *addProposers* is $50180 + (23586 * n_c)$ which is $O(n_c)$. The measured gas consumption of *proposer* and the *toProposalState* are $O(1)$ and fairly low. *addVoters*'s gas cost is directly related with the voter count. Ethereum has a block gas capacity which changes between 15 and 20 million gas. We were able to add 10,000 voter IDs in a single transaction for *addVoters* function call, without exceeding the block limit. In our tests, we found that *addVoters* call exceeds this limit with approximately 30,000 voters. However, as mentioned before, this is only for a single call. Election Authority can issue multiple calls to keep adding new IDs. For example 100,000 voters can be added by splitting vodeIDs into 10 different *addVoters* calls with 10,000 voters for each. We have mentioned that the *addVoters* function can be offloaded to the election authority with an additional trust in the authority. We have conducted a test for this version of the *addVoters* function with candidateCount=10 and voterCount=10,000. The modified function consumes only 70,219 gas as it only registers the Merkle tree root without publishing events for identity commitments.

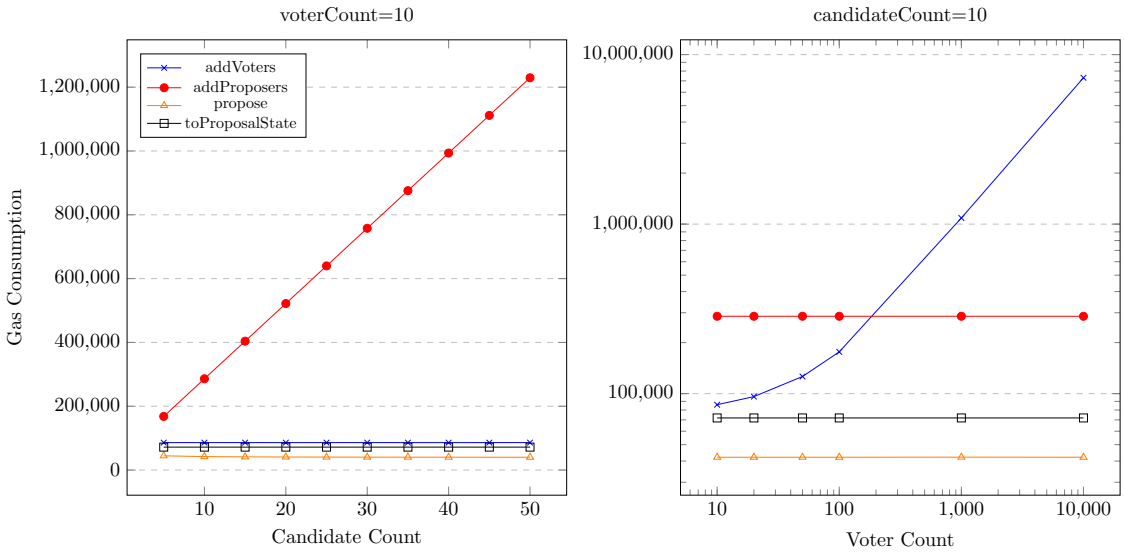

Figure 5.2. Election Authority Functions.

We also run tests for Merkle tree generation with different voter counts. Merkle trees must be generated by both election authority and voters. Election Authority must generate the Merkle tree and register the tree root to the smart contract. Voters also need to generate the tree to find their path elements and indexes for Merkle proofs. We analyzed all these tree functions *genTree*, *genPathElementsAndIndex*, *getRoot* with increased voter count. Figure 5.3 shows the result. The tree file size represents Merkle proof file size. The file contains all Merkle proofs for every individual leaf. It means that voters can grab the file and find their related Merkle proofs without actually generating the tree. Merkle tree and proof generation have a linear relation with the voter count. We found that the tree can be constructed within 42 minutes for 100,000 voters. Merkle proof generation takes only 6 seconds for 100,000 voters. The generated file size takes 165,5 megabytes for 100,000 voters, which makes it practical enough to be shared online.

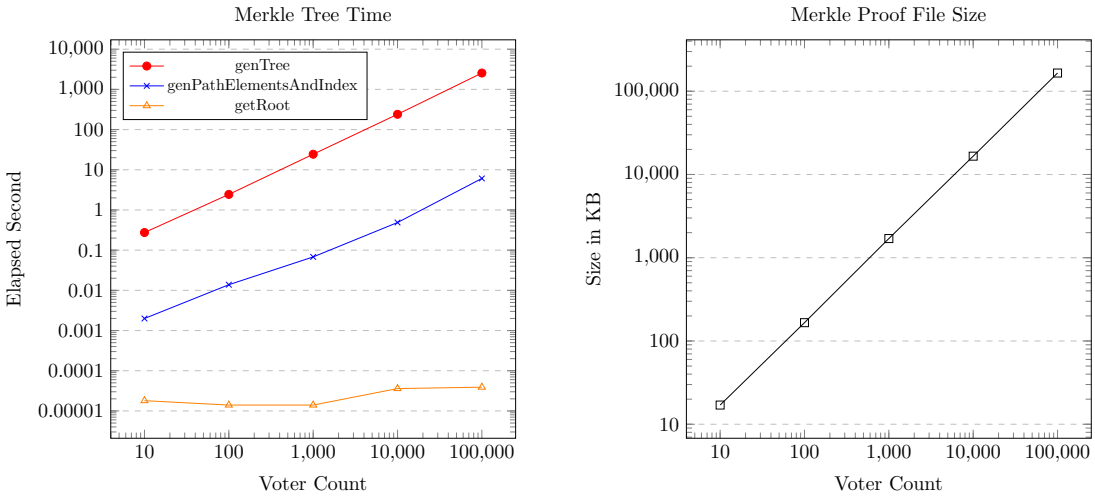

Figure 5.3. Merkle Tree Functions.

We also analyzed the Semaphore circuit setup times, file sizes, and the witness & proof generation times. Results can be found in Table 5.1. Results show that compiling the circuit and generating keys take a total of almost 15 minutes and 250-megabyte file size. This is fairly feasible since it is a one-time setup only. Each voter uses *genID*, *genIDCommit* to generate their identity and identity commitments. In the table, it can be seen that these functions take sub-second times. Voters use *genWitness* and *genProof* in conjunction to generate their zero-knowledge proofs. They take around 10 secs to generate proof for the Semaphore verifier.

Table 5.1. Semaphore Circuit & Function Times.

Operation	Time (sec)	File Size (mb)
Compile Circuit	206	132
Key Generation	703	128
Generate Verifier Contract	0.39	0.01

Function	Time (sec)
genID	0.027504
genIDCommit	0.099805
genWitness	1.622
genProof	8.446

5.2.1. Merkle Forest

We also run tests for the Merkle forest extension. We have deployed several smart contracts with various fixed-size trees and added voters with *addVoters*. We show results for two different implementations in Table 5.2. One of the implementations takes the keccak hash function; the other one uses public inputs. It is obvious that implementation with *keccak* hash function consumes less gas for increased voter count. It is due to the fact that the verifier in the implementation with the keccak hash function verifies only a single hash input. Whereas in the other implementation, all leaves are passed to the verifier to be verified. Passing a full list instead of a single hash increases the gas consumption with respect to increased voter count. However, the implementation without *keccak* hash function produces fewer constraints. As a result, generation times and file sizes are lower. We think that the circuit with keccak is more feasible for a smart contract approach since it consumes almost one-quarter of gas compared to the one without keccak. As we told before, we assume the election authority has enough resources, i.e., enough to cover gas prices and computational resources to generate zero-knowledge proofs. Inserting a 256-sized tree with a keccak circuit consumes 482,409 gas, whereas the original implementation without Merkle forest consumes 276,929 for 200 voter registration. The Merkle forest solution becomes more valuable for small-scale elections as the solution reduces the trust assumption in the election authority without consuming too much gas. It is also still possible to generate a 256-size tree circuit and insert multiple trees to the smart contract, thus increasing the total voter count with several rounds. However, increasing the fixed tree size in circuits certainly helps to reduce the gas cost for a single leaf insertion to the smart contract. The insertion/size gas ratio can be seen in in Table 5.2 column *Insert/Size Gas*. For large-scale elections, we think this solution is not feasible as both gas consumption and circuit costs (file sizes, generation times) increase dramatically with increased tree sizes. However, Merkle Forest extension can offer a viable solution for small/medium scale elections.

Table 5.2. Test Results for Merkle Forest Implementations.

Keccak	Size	Constraints	Compile Time	Setup Time	Witness Time	Proof Time	Deploy Gas	Insert Gas	Insert Gas per Size	Compiled Size (MB)	Proving Key Size (MB)
Yes	2	156,423	0:00:05	0:01:14	0:00:02	0:00:06	1,362,455	312,823	156,411.5	52	55
No	2	2641	0:00:01	0:00:01	0:00:00	0:00:01	1,310,489	297,325	148,662.5	12	0.9
Yes	4	163,237	0:00:06	0:01:19	0:00:02	0:00:07	1,400,793	315,557	78,889.25	86	57
No	4	7921	0:00:02	0:00:04	0:00:02	0:00:01	1,398,693	315,519	78,879.75	46	2.4
Yes	8	329,499	0:00:13	0:02:23	0:00:04	0:00:14	1,404,040	316,799	39,599.875	205	114
No	8	18,481	0:00:05	0:00:10	0:00:03	0:00:05	1,508,133	349,425	43,678.125	125	6.5
Yes	16	661,908	0:00:29	0:04:44	0:00:10	0:00:30	1,404,004	322,106	20,131.625	452	229
No	16	39,601	0:00:12	0:00:24	0:00:07	0:00:11	1,714,373	419,031	26,189.4375	292	14
Yes	32	1,326,724	0:01:05	0:09:42	0:00:22	0:01:07	1,405,936	332,534	10,391.6875	957	459
No	32	81,874	0:00:27	0:00:53	0:00:16	0:00:25	2,128,664	558,448	17,451.5	636	28
Yes	64	2,656,356	0:02:53	0:21:06	0:00:53	0:02:28	1,404,004	353,931	5530.171875	1900	918
No	64	166,321	0:01:02	0:01:52	0:00:34	0:00:53	2,948,257	839,127	13,111.35938	1300	57
Yes	128	5,162,019	0:05:18	0:37:37	0:01:36	0:04:29	1,404,016	396,685	3099.101563	3900	1800
No	128	335,281	0:02:16	0:03:17	0:01:08	0:01:48	4,595,134	1,405,349	10,979.28906	2700	114
Yes	256	10,173,347	0:30:49	1:16:15	0:03:17	0:10:03	1,404,640	482,409	1884.410156	7800	3500
No	256	673,201	0:07:56	0:06:48	0:02:17	0:03:55	7,890,686	2,559,015	9996.152344	5400	228

5.2.2. Evaluation

We compared the ElectAnon with other previous similar works like McCorry et al. [41] and PriScore [37]. McCorry et al. [41] has only a single *Yes/No* choice system, whereas PriScore [37] uses a score-based ranked-choice election with multiple candidates. McCorry et al. [41] conducted their gas consumption tests with 40 voters. PriScore [37] also shows gas costs for each of their different functions. We computed their total gas costs for 40 voters and ten candidates. We also run our tests with 40 voters and ten candidates. A comparison table can be found in Table 5.3. We found that ElectAnon offers an 83% and 89% decrease in total election gas costs in comparison with the McCorry et al. [41] and Priscore [37] respectively.

Table 5.3. Gas Comparison Table (voterCount=40, candidateCount=10).

Entity: Transaction	McCorry et al. [41]	Priscore [37]	This Work
A: Deploy	6,215,811	-	3,430,754
A: Add Voters	2,153,461	-	113,963
A: Add Proposers	-	-	286,040
A: State Change	3,320,433	-	71,877
P: Propose	-	-	42,681
V: Register	763,118	-	-
V: Commit	70,112	1,107,374	312,856
V: Vote	2,490,412	3,579,468	105,140
A: Tally	746,485	60,096	48,937
Authority Total	12,436,190	60,096	3,665,531
Proposer Total	-	-	42,681
Voter Total	3,323,642	4,686,842	417,996
Election Total	145,381,870	187,533,776	20,812,181

At the time of writing this work, the gas price in Ethereum nearly 100 *gwei*, and one ETH is approximately 4,500 $. A gwei equals to 10^9 ETH, so a gwei roughly equals to $4,500/10^9 = 0.0000045$$. It means that a single gas costs $100*0.0000045 = 0.00045$$. In our work, voters total gas cost is 417,996 which makes $0.00045*417,996 = 188.0982$$. We considered running the ElectAnon in another Ethereum Virtual-Machine compatible network, *Avalanche*. Avalanche implements a novel consensus mechanism with proof-of-stake Sybil protection [73]. It offers a faster finalization time and increased transaction per second rate with very low gas fees when compared to Ethereum. The Avalanche gas prices change between 25-150 *nAVAX* (equivalent to *gwei)* [74]. The current price of Avalanche is 85$ on average. If we take the gas price as nearly 100 nAVAX, then our voting transactions costs would be equivalent to $100/10^9 * 85 * 417,996 = 3.552966$$. We run an election with ten voters and ten candidates in Avalanche local network. We found that ElectAnon is compatible with the Avalanche network and it

costs the same gas as the Ethereum network. However, since gas prices and the AVAX price is much lower than the Ethereum, it reduces the cost of election significantly.

In our tests, we found that the voter functions, i.e *commitVote*, *revealVote*, *electionResult* are not affected by the increased voter count. It means that voters do not pay for extra gas in case of a large-scale election. The cost of election authority increases with both candidate count for *addProposers* and voter count for *addVoters*. We assume that the election authority has enough resources to conduct the election.

A possible bottleneck of ElectAnon could be adding voters to the eligible list. It costs almost a total of 10,000,000 gas for 10,000 voter registration which almost costs $5000 USD in Ethereum. It means that a single leaf insertion costs approximately $5000/10,000 = \$0.5$ USD. This is a fair cost for small to medium-scale elections, i.e., up to 10,000 voters. The gas cost increases linearly. A large-scale election with 1,000,000 voters would cost approximately $500.000 USD. Even though our work offers the best gas consumption amongst others like McCorry et al. [41] and Priscore [37], $500.000 USD is still too much. The total USD cost can be decreased by using *Avalanche* network, which has significantly less gas price, as mentioned in the previous paragraph. The same total cost to register 1,000,000 voters can be decreased to almost $100/10^9 * 85 * 10,000,000 * 100 = 8500\$$ USD in Avalanche.

We also discussed a potential solution to reduce the cost of registering voters with *Assisted Merkle Trees* in Section 4.4.2. We found that the potential solution registers 100,000 voters with only 70,219 gas costs which are approximately $30 USD in Ethereum Network. The Merkle tree file takes almost 100 mega-bytes for 100,000 tree leaves. We used no compression or encoding in our experiments. This size potentially can be decreased further with compression. The potential solution can offer a feasible way to share Merkle trees via cloud or IPFS.

6. CONCLUSION

6.1. Remarks

In this work, we proposed a blockchain-based, anonymous ranked-choice voting protocol. Our work, ElectAnon, ensures full anonymity with zero-knowledge proofs. We used a zero-knowledge gadget, *Semaphore* [15], which provides anonymous membership proofs with the efficient zk-SNARK technique. Our protocol also provides efficient mechanisms for ranked-choice voting. We used an effective algorithm [62] to encode and decode our ranked-choice lists with integers. ElectAnon is designed to be fully robust and uninterruptible in the voting phase. We implemented the protocol in Ethereum smart contracts to ensure decentralization and robustness. Detailed implementation and analysis of the protocol are presented in work with technical discussions. We also defined some of the most critical election requirements and provided a deep-down analysis of prior works. We also discussed and analyzed alternative extensions like *Merkle Forests*, *Tideman method* and *Multiple Elections* for our protocol. We have run some real-world experiments and shown the results. We also compared our work with prior works. Our protocol not only assures critical election requirements but also scales to be used in large-scale elections. We think that ElectAnon can also be beneficial for governance applications like decentralized autonomous organizations (DAOs).

6.2. Future Work

Currently, ElectAnon returns only the first winners of the elections. The whole tallying information is kept available on the smart contract after tallying phase ends, so showing remaining results can be achieved through very little modification. We left the implementation for this as future work.

We assumed that election authority establishes a secure channel for voters and proposers. We left the actual implementation of this channel as future work. Recall

that we also mentioned it is possible to increase the election authority number so the authority role can be more decentralized and voter registration costs can be shared. We also left the actual implementation of working with multiple election authorities as future work.

Our current voting scheme expects voters to form a full list of candidates. Because of that, each voter has to decide their preferences for all candidates. In other words, each preference list must contain all possible candidates in different orders. This can be a burden for elections with many candidates, as voters have to evaluate all candidates and form a list of all candidates sorted by their preferences. This can be avoided by specifying a smaller preference list size. For example, if there are 50 candidates, instead of forming a permutation of 50 candidates, the election can require voters to decide on some smaller number of candidates like 10 or 5. So voters can evaluate a smaller candidate subset instead of the whole set. Authors of the *Ranking and unranking permutations in linear time* [62] mention a possible extension to accomplish k-permutations of an n-set. Our preference list encoding can use this approach to reduce the list size, and thus both usability and gas efficiency can be increased.

REFERENCES

1. Gibson, J. P., R. Krimmer, V. Teague and J. Pomares, "A Review of E-voting: the Past, Present and Future", *Annals of Telecommunications*, Vol. 71, No. 7, pp. 279–286, 2016.

2. Al-Janabi, S. and N. Hamad, "Security of Internet Voting Schemes: A Survey", *REVISTA AUS Journal, Special Issue*, Vol. 26, No. 2, pp. 260–270, 2019.

3. Alvarez, R. M., T. E. Hall and A. H. Trechsel, "Internet Voting in Comparative Perspective: the Case of Estonia", *PS: Political Science & Politics*, Vol. 42, No. 3, pp. 497–505, 2009.

4. Serdult, U., M. Germann, F. Mendez, A. Portenier and C. Wellig, "Fifteen Years of Internet Voting in Switzerland [History, Governance and Use]", *2015 Second International Conference on eDemocracy eGovernment (ICEDEG)*, pp. 126–132, 2015.

5. SCOOP4C, *Estonian Internet Voting*, `"https://scoop4c.eu/cases/estonian-internet-voting"`, accessed in November 2021.

6. Desilver, D., "Mail-in Voting Became Much More Common in 2020 Primaries as COVID-19 Spread", *Pew Research Center*, 2020.

7. Krimmer, R., D. Duenas-Cid and I. Krivonosova, "Debate: Safeguarding Democracy During Pandemics. Social Distancing, Postal, or Internet Voting—the Good, the Bad or the Ugly?", *Public Money & Management*, Vol. 41, No. 1, pp. 8–10, 2021.

8. James, T. S., "New Development: Running Elections During a Pandemic", *Public Money & Management*, Vol. 41, No. 1, pp. 65–68, 2021.

9. Sachdeva, M., G. Singh, K. Kumar and K. Singh, "A Comprehensive Survey of Distributed Defense Techniques Against DDoS attacks", *International Journal of Computer Science and Network Security*, Vol. 9, No. 12, pp. 7–15, 2009.

10. Agre, P. E., "P2P and the Promise of Internet Equality", *Communications of the ACM*, Vol. 46, No. 2, p. 39–42, 2003.

11. Hjálmarsson, F. T., G. K. Hreiðarsson, M. Hamdaqa and G. Hjálmtýsson, "Blockchain-Based E-Voting System", *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 983–986, 2018.

12. Taş, R. and Ö. Ö. Tanrıöver, "A Systematic Review of Challenges and Opportunities of Blockchain for E-Voting", *Symmetry*, Vol. 12, No. 8, 2020.

13. El Faqir, Y., J. Arroyo and S. Hassan, "An Overview of Decentralized Autonomous Organizations on the Blockchain", *Proceedings of the 16th International Symposium on Open Collaboration*, OpenSym 2020, Association for Computing Machinery, New York, NY, USA, 2020.

14. Buterin, V., *A Next-Generation Smart Contract and Decentralized Application Platform*, Tech. rep., Ethereum Foundation, 2014.

15. Gurkan, K. and K. W. Jie, *Community Proposal: Semaphore: Zero-Knowledge Signaling on Ethereum*, Tech. rep., ZKProof Standards, 2020.

16. Sarmah, S. S., "Understanding blockchain technology", *Computer Science and Engineering*, Vol. 8, No. 2, pp. 23–29, 2018.

17. Nakamoto, S., *Bitcoin: A Peer-to-Peer Electronic Cash System*, Tech. rep., Bitcoin, 2008.

18. Pierro, G. A., R. Tonelli and M. Marchesi, "An Organized Repository of Ethereum Smart Contracts' Source Codes and Metrics", *Future Internet*, Vol. 12, No. 11,

2020.

19. *Solidity 0.8.7 Documentation*, https://docs.soliditylang.org/en/v0.8.7/, accessed in November 2021.

20. Kevin Sekniqi, S. B., Daniel Laine and E. G. Sirer, *Avalanche Platform Whitepaper*, Tech. rep., AvaLabs, 2020.

21. Binance, "Binance Chain Community Releases Whitepaper for Enabling Smart Contracts", *Binance Blog*, 2020.

22. Barinov, I. I., V. Arasev, A. Fackler, V. Komendantskiy, A. Gross, A. Kolotov and D. Isakova, "POSDAO: Proof of Stake Decentralized Autonomous Organization", *Applied Computing eJournal*, 2019.

23. Ethereum Foundation, *Istanbul - October 2019 Planned Ethereum Network Upgrade*, 2019, https://eth.wiki/roadmap/istanbul, accessed in November 2021.

24. Kumar, A., "Ownerless Ownership, Trustless Trust — DAOs, the Future of Governance", *BeInCrypto*, 2021.

25. Meir, R., "Plurality Voting Under Uncertainty", *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29, 2015.

26. Bormann, N.-C. and M. Golder, "Democratic Electoral Systems around the world, 1946–2011", *Electoral Studies*, Vol. 32, No. 2, pp. 360–369, 2013.

27. ACE, *The Global Distribution of Electoral Systems*, https://aceproject.org/main/english/es/esh.htm, accessed in November 2021.

28. Brogan, D. W., "Political Parties: Their Organization and Activity in the Modern State. By Maurice Duverger. Translated by Barbara and Robert North. (New York: John Wiley & Sons, Inc.1954. Pp. xxxvii, 439.)", *American Political Science*

Review, Vol. 49, No. 3, p. 889–890, 1955.

29. Herron, M. and J. Lewis, "Did Ralph Nader Spoil a Gore Presidency? A Ballot-Level Study of Green and Reform Party Voters in the 2000 Presidential Election", *Quarterly Journal of Political Science*, Vol. 2, No. 3, pp. 205–226, 2007.

30. Anest, J., "Ranked Choice Voting", *Journal of Integral Theory and Practice*, Vol. 4, No. 3, pp. 23–40, 2009.

31. Diorio, D. and W. Underhill, "Ranked-Choice Voting", *LegisBriefs*, Vol. 25, No. 24, 2017.

32. Gritzalis, D. A., "Principles and Requirements for a Secure E-voting System", *Computers & Security*, Vol. 21, No. 6, pp. 539–556, 2002.

33. Jafar, U., M. J. A. Aziz and Z. Shukur, "Blockchain for Electronic Voting System—Review and Open Research Challenges", *Sensors*, Vol. 21, No. 17, 2021.

34. Hussien, H. and H. Aboelnaga, "Design of a Secured E-voting System", *2013 International Conference on Computer Applications Technology (ICCAT)*, pp. 1–5, 2013.

35. Yang, X., X. Yi, S. Nepal, A. Kelarev and F. Han, "Blockchain Voting: Publicly Verifiable Online Voting Protocol Without Trusted Tallying Authorities", *Future Generation Computer Systems*, Vol. 112, pp. 859–874, 2020.

36. Zhang, S., L. Wang and H. Xiong, "Chaintegrity: Blockchain-enabled Large-scale E-voting System With Robustness and Universal Verifiability", *International Journal of Information Security*, Vol. 19, pp. 323–341, 2019.

37. Yang, Y., Z. Guan, Z. Wan, J. Weng, H. H. Pang and R. H. Deng, "PriScore: Blockchain-Based Self-Tallying Election System Supporting Score Voting", *IEEE Transactions on Information Forensics and Security*, Vol. 16, pp. 4705–4720, 2021.

38. Merener, M. M., "Theoretical Results on De-Anonymization via Linkage Attacks", *Transactions on Data Privacy*, Vol. 5, No. 2, p. 377–402, 2012.

39. Shirazi, F., S. Neumann, I. Ciolacu and M. Volkamer, "Robust Electronic Voting: Introducing Robustness in Civitas", *2011 International Workshop on Requirements Engineering for Electronic Voting Systems*, pp. 47 – 55, 2011.

40. Pawlak, M. and A. Poniszewska-Marańda, "Trends in Blockchain-based Electronic Voting Systems", *Information Processing & Management*, Vol. 58, No. 4, p. 102595, 2021.

41. McCorry, P., S. F. Shahandashti and F. Hao, "A Smart Contract for Boardroom Voting with Maximum Voter Privacy", A. Kiayias (Editor), *Financial Cryptography and Data Security*, pp. 357–375, Springer International Publishing, 2017.

42. Panja, S., S. Bag, F. Hao and B. Roy, "A Smart Contract System for Decentralized Borda Count Voting", *IEEE Transactions on Engineering Management*, Vol. 67, No. 4, pp. 1323–1339, 2020.

43. Hao, F., P. Ryan and P. Zielinski, "Anonymous Voting by Two-round Public Discussion", *Information Security, IET*, Vol. 4, pp. 62 – 67, 2010.

44. etherscan.io, *Ethereum Average Block Size Chart*, `http://etherscan.io/chart/blocksize`, accessed in October 2021.

45. Brandt, F., "Efficient Cryptographic Protocol Design Based on Distributed El Gamal Encryption", D. H. Won and S. Kim (Editors), *Information Security and Cryptology - ICISC 2005*, pp. 32–47, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

46. Linoy, S., N. Stakhanova and S. Ray, "De-Anonymizing Ethereum Blockchain Smart Contracts through Code Attribution", *International Journal of Network Management*, Vol. 31, No. 1, 2021.

47. Goldwasser, S., S. Micali and C. Rackoff, "The Knowledge Complexity of Interactive Proof-Systems", *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, p. 291–304, Association for Computing Machinery, New York, NY, USA, 1985.

48. Petkus, M., "Why and How zk-SNARK Works", *arXiv*, Vol. abs/1906.07221, 2019.

49. Morais, E., T. Koens, C. van Wijk and A. Koren, "A Survey on Zero Knowledge Range Proofs and Applications", *SN Applied Sciences*, Vol. 1, No. 8, p. 946, 2019.

50. Company, E. C., *What are zk-SNARKs?*, `https://z.cash/technology/zksnarks/`, accessed in November 2021.

51. Aztec, *Aztec Network*, `https://aztec.network/`, accessed in November 2021.

52. Starkware, *StarkNet*, `https://starkware.co/starknet/`, accessed in November 2021.

53. Partala, J., T. H. Nguyen and S. Pirttikangas, "Non-Interactive Zero-Knowledge for Blockchain: A Survey", *IEEE Access*, Vol. 8, pp. 227945–227961, 2020.

54. iden3, *Circom 2 Documentation*, `https://docs.circom.io/`, accessed in November 2021.

55. ZoKrates, *Introduction - ZoKrates*, `https://zokrates.github.io/`, accessed in November 2021.

56. Jie, K. W., "To Mixers and Beyond: Presenting Semaphore, a Privacy Gadget Built on Ethereum", *Coinmonks*, 2020.

57. appliedzkp, *libsemaphore*, `https://semaphore.appliedzkp.org/libsemaphore.html`, accessed in November 2021.

58. Company, E. C., *Parameter Generation*, `https://z.cash/technology/paramgen/`, accessed in October 2021.

59. Jie, K. W., "Announcing the Perpetual Powers of Tau Ceremony to Benefit all zk-SNARK Projects", *Coinmonks*, 2020.

60. Jie, K. W., *Perpetual Powers of Tau*, `https://github.com/weijiekoh/perpetualpowersoftau`, accessed in October 2021.

61. Jie, K. W., "Restarting the Semaphore Random Value Generation process", *Medium*, 2020.

62. Myrvold, W. and F. Ruskey, "Ranking and Unranking Permutations in Linear Time", *Information Processing Letters*, Vol. 79, No. 6, pp. 281–284, 2001.

63. Tideman, T. N., "Independence of Clones as a Criterion for Voting Rules", *Social Choice and Welfare*, Vol. 4, No. 3, pp. 185–206, 1987.

64. Benet, J., "IPFS - Content Addressed, Versioned, P2P File System", *arXiv*, Vol. abs/1407.3561, 2014.

65. Emerson, P., "The Original Borda Count and Partial Voting", *Social Choice and Welfare*, Vol. 40, No. 2, pp. 353–358, 2013.

66. Bassett, G. W. and J. Persky, "Robust Voting", *Public Choice*, Vol. 99, No. 3, pp. 299–310, 1999.

67. Sayeed, S., H. Marco-Gisbert and T. Caira, "Smart Contract: Attacks and Protections", *IEEE Access*, Vol. PP, pp. 1–1, 2020.

68. Chow, J., "A Guide to Events and Logs in Ethereum Smart Contracts", *ConsenSys*, 2016.

69. Albrecht, M., L. Grassi, C. Rechberger, A. Roy and T. Tiessen, "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity", J. H. Cheon and T. Takagi (Editors), *Advances in Cryptology – ASIACRYPT 2016*, pp. 191–219, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

70. Jie, K. W., *Gas and Circuit Constraint Benchmarks of Binary and Quinary Incremental Merkle Trees Using the Poseidon Hash Function*, 2020, `https://ethresear.ch/t/gas-and-circuit-constraint-benchmarks-of-binary-and-quinary-incremental-merkle-trees-using-the-poseidon-hash-function/7446`, accessed in October 2021.

71. Solidity, *Contract ABI Specification — Solidity 0.8.7 Documentation*, `https://docs.soliditylang.org/en/v0.8.7/abi-spec.html`, accessed in October 2021.

72. Nomic Labs, *Hardhat*, `https://hardhat.org/`, accessed in November 2021.

73. Rocket, T., M. Yin, K. Sekniqi, R. van Renesse and E. G. Sirer, "Scalable and Probabilistic Leaderless BFT Consensus through Metastability", *arXiv*, Vol. abs/1906.08936, 2020.

74. Ava Labs, *Avalanche Transaction Fee*, `https://docs.avax.network/learn/platform-overview/transaction-fees`, accessed in November 2021.

APPENDIX A: APPENDIX

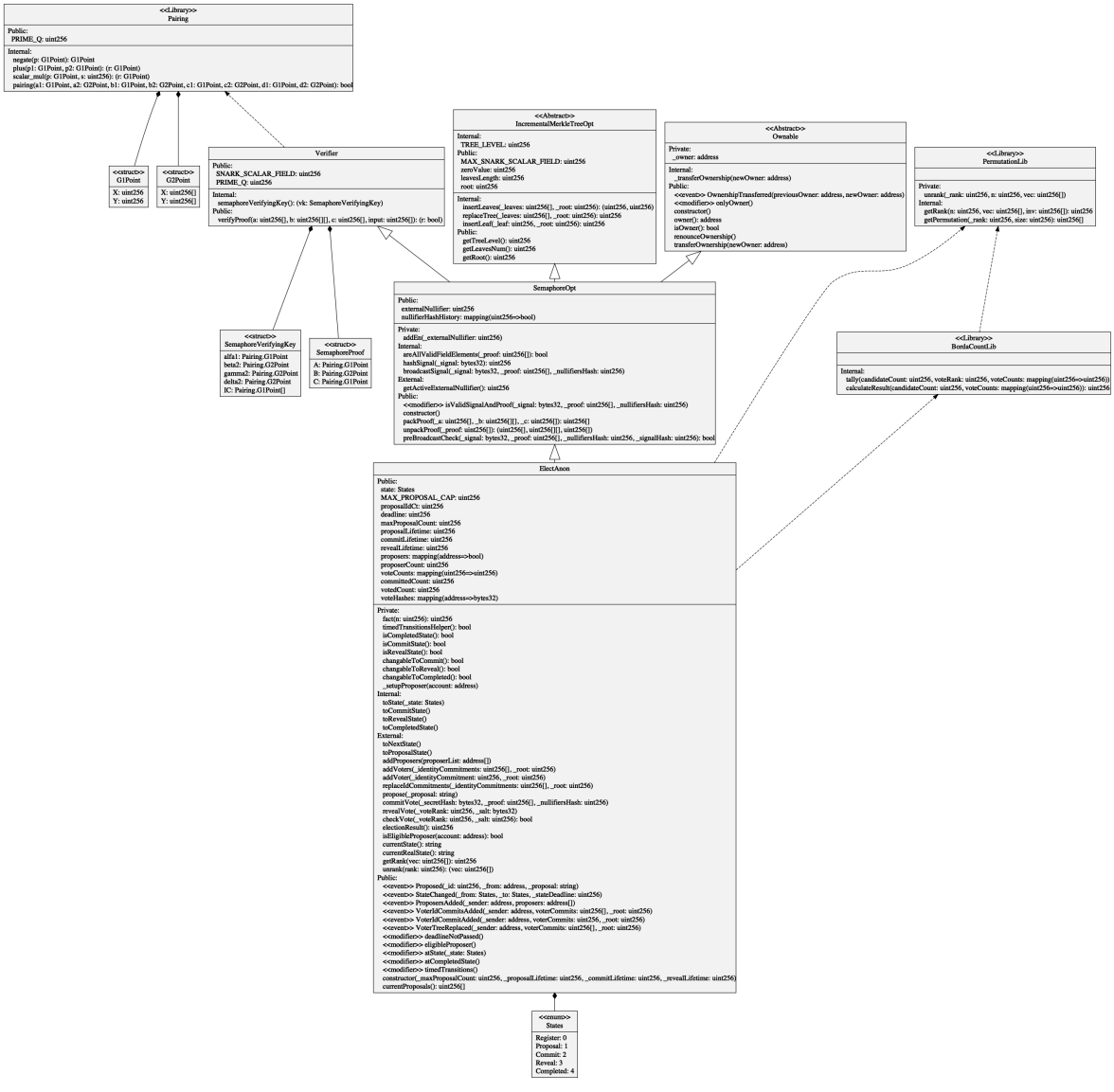

Figure A.1. ElectAnon Class Diagram.